Syllabus

lunedì, luglio 17, 2017
5:05

- Introduction
  - Notion of concurrency
  - Partial and total ordering of activities
  - Processes and threads
    - Creation of Java threads
    - Thread pooling and Executors for Java tasks
    - Use of Future objects with ExecutorService
  - Speedup of a parallelized application: Amdhal's law, Gustafson's law
  - Models and paradigms: Shared memory vs. message passing
- Shared memory model
  - Mutual exclusion
    - Classes of critical sections
    - Naif software approaches, Peterson's solution, the Bakery algorithm
    - The Lock concept and object-oriented idioms
    - Hardware support by Test-And-Set, Compare-And-Swap, Fetch-And-Add
  - Synchronization/communication
    - Locks
      - Mutexes and spinlocks
      - Avoidance of busy-waiting and starvation, 'sleeping' locks
      - Two-phase locks
      - Explicit locks in Java
    - Higher-level control: Condition variables
    - The Monitor construct
      - Rationale for the Construct
      - Hoare's Monitor, semantics 'signal-and-urgent-wait', 'signal-and-wait', 'signal-and-return'
      - Mesa-style Monitor and Java Monitor; notifyAll()
    - Semaphores
      - Definition
      - Use of counting semaphores in Java
    - Memory consistency
      - Models, and memory barriers
      - The Java Memory Model based on happened-before
  - Typical synchronization problems
    - Bounded buffer - Java BlockingQueue
    - Readers and writers - Java ReadWriteLock
    - Dining philosophers
    - Deadlock and livelock
      - Coffman's conditions
      - Wait-for graph
      - Approaches to deadlock management, trylock
  - Non-blocking algorithms
    - Generalities, use of CAS, optimistic assumptions
    - Treiber's stack
    - Michael and Scott's queue
  - Development of Java multithreaded applications
    - Synchronized/concurrent collections
    - Synchronizers (latches, etc.)
    - Unit testing, generalities on testing of concurrent Java programs

- Message-passing model
  - Models for distributed systems
  - Asynchronous and synchronous models
    - Process addressing and channels
    - Unblocking and blocking constructs
    - Guarded commands
    - Bounded buffer problem in asynchronous model
  - The Actor model
    - General characteristics
    - Relation to multithreaded and distributed applications
    - Example: the Akka framework
  - Precedence and causality
    - Happened before relation
    - Lamport's timestamps (partial/total order)
    - Vector timestamps, properties, consistent cuts
    - Physical clock sychronization - concepts and algorithms
  - Distributed mutual exclusion
    - Centralized solution
    - Ricart-Agrawala algorithm, token ring solution
  - Dealing with faults
    - Fault models
    - Election: ring-based and bully algorithm
    - Basic multicast
    - Consensus
    - The Byzantine Generals problem
      - Case with N=3
      - Possibility of solution for N>=3f+1
    - FLP restriction
- Communication APIs
  - Low-level APIs
    - MPI basic concepts
    - Java networking classes
      - Connectionless and connection-oriented communication
  - Direct and indirect communication
    - RPC - Remote Procedure Call
      - Request-reply protocols, idempotent operations
      - RPC's Interfac Description Languages (IDLs)
      - Call semantics
      - RPC implemetation
    - RMI - Remote Method Invocation
      - Remote interfaces and remote references
      - Architecture: servants and location services
      - Distributed garbage collection
      - Remote class loading
    - Message-oriented middleware
      - Message queues and publish-subscribe systems
      - JMS - Java Message Service
        - JMS providers
        - JMS programming model, connections, and sessions

- Frameworks for applications' support
  - Basic patterns and architectures
    - 3-tiers architecture
    - Container pattern and interception
    - MVC - Model View Controller
  - Web applications
    - Architecture of a Web server
    - State-aware computations and session management
    - Java Servlets
      - Lifecycle and service methods
      - Classes, configuration, and deployment
      - Concurrency issues
      - Servlet filters
    - Deployment descriptors, convention over configuration
  - Distributed components
    - Application servers
    - EJBs - Enterprise Java Beans
      - Session EJBs: stateless and stateful
      - Management strategies and implementation
      - Pooling and passivation
      - EJB interfaces and lookup via JNDI
      - Context and Dependency Injection via annotations

Access control modifiers

Tuesday, July 18, 2017
18:03

Modifier	Accessible from (in addition to defining class)
public	Any class of any package
protected	Subclasses + any class of the same package
private	**Only classes that share the same top-level class as the defining class**
default	Any class of the same package

Nested classes

Tuesday, July 18, 2017
17:48

Summary

Type	Inner (associated with outer instance)	Definition point	Visibility
Static nested class	No	As a member of another class.	Depends on access modifier.
Non-static member class (regular inner class)	Yes	As a member of another class.	Depends on access modifier.
Local class	Yes (if defined in non-static method)	Inside a method.	From the point it is defined to the end of the method.
Anonymous class	Yes (if defined in non-static method)	As an expression (since defining it also returns the instance).	None.

Types

- Static

```
class A {
        ...
        static class B {
                ...
        }
        ...
}
```

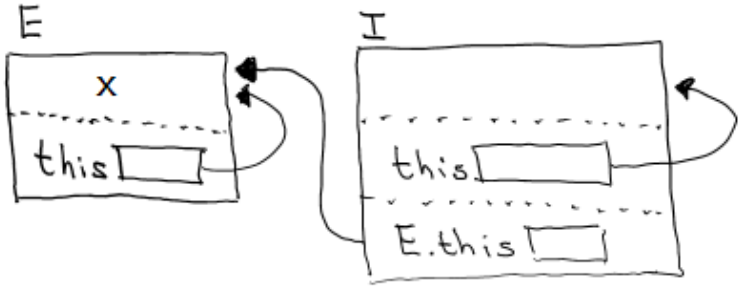
  - **Indipendent**: can be instantiated without creating the outer class
  - Can **only access static members** of the enclosing class
  - Can have any visibility
- Inner
  - An instance of the inner class is **associated with an instance of the outer one**
    - The inner one **can access non-static members** of the outer one
  - The inner class instance has only one pointer to the outer class instance
    - One outer class can be referred by multiple inner class instances
  - Non-static member (the default "inner class")

```
class A {
        ...
        class B {
                ...
        }
        ...
}
```

    - The inner class can be accessed by A.B
    - It can have any visibility
    - Construction

```
A a = new A();
A.B ab = a.new B();
```

- Accessing the enclosing class




```
class A {
	void foo() { System.out.println("A"); }
	class B {
		void foo() { System.out.println("B"); }
		void bar() { A.this.foo(); }  //prints A
	}
}
```

- Local
  - Declared inside a method
  - **Two types**: behaves differently depending on where it's defined

```
class Outer {
    private int x;

    // 1) Inside a non-static method: inner class
    void f() {
        final int y = 4;

        // Visible from here 'till the end of the method
        class Inner {
            void inc() {
                x += y; // LEGAL: it can see y 'cause it
            }
        }

        Inner i = new Inner();
    }

    // 2) Inside a static method: static nested class
    static void g() {
        Inner i = new Inner(); // ILLEGAL: class declara

        // Visible from here 'till the end of the method
        class Inner { // LEGAL: it can use the same name
            void inc() {
                x++; // ILLEGAL: x is not static
            }
        }
    }
}
```

<<ilc.java>>

- Anonymous
  - Declared inside a method, without naming the class
  - Example

```
interface A {
    void g();
}

class B {
    void f() {
        A a = new A() {
            public void g() {}
        };

        a.g();
    }
}
```

  - Constructors

```
class A {
    private int a;

    public A(int x) { a = x; }
    void g() { System.out.println("What's not printed.")
}

class B {
    void f() {
        (new A(3) { void g() { System.out.println("What'
    }
}

class Main {
    public static void main(String[] args) {
        B b = new B();
        b.f();
    }
}
```

<<ac.java>>

- **Shell**: reset && javac ac.java && java Main
- **Output**: What's printed

Concurrency

Wednesday, July 19, 2017
17:56

Definitions

- Concurrency: two or more activities carried out at the same time

- Distribution: different activities are carried out concurrently on different machines that communicate through a network

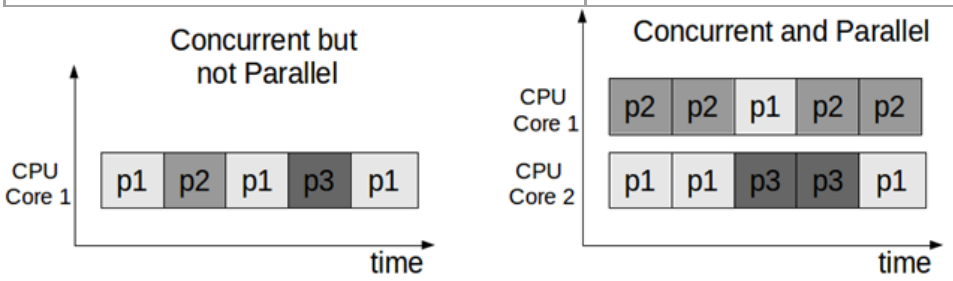
Precedence graph	**Partially Ordered SET (POSET)**
Graph that shows the order of activities. • Job repartition to workers can be done by dividing the precedence graph in subsets of adjacent activities.	Set of activies, in which the **partial order relationship** ($\leq$) is defined. • Total order relationship When a partial order exists for each pair of activities.

Concurrent activities

Two activities i, j are concurrent iff they are not partially ordered: $\neg(i \leq j), \neg(j \leq i) \Leftrightarrow i \parallel j$

Different types of concurrency

True concurrency	[Interleaving] concurrency



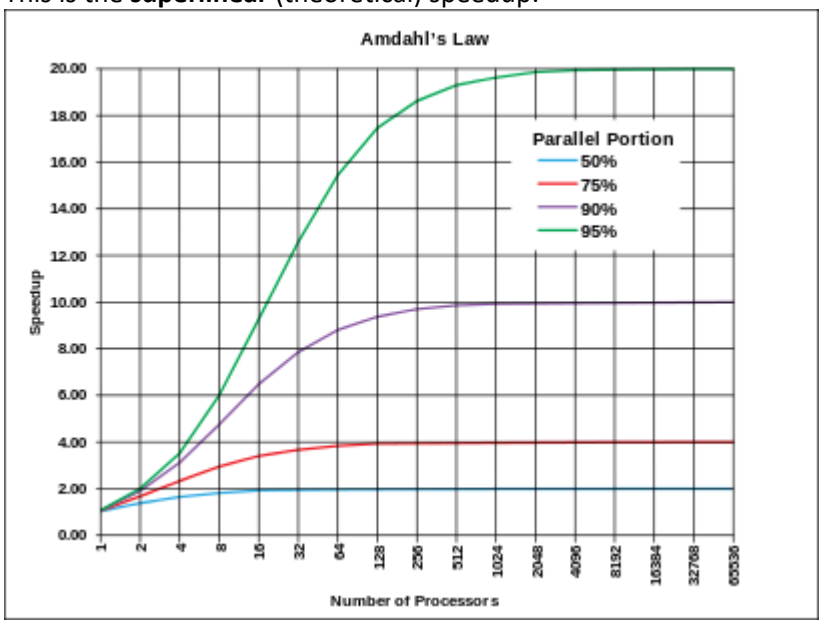
Parallelized programs' speedup

- Two laws for the same thing
  - Amdahl's law
  - Gustafson's law
- CPU clock cannot be increased without limits due to physical constraints
  - That's why parallelism is exploited
- Notation
  - P: activity
  - $T_i(P)$: time required to complete P with i workers executing in parallel.
    - $T_i(P) = s + p/i$
      - s: time required to execute the sequential portion.
        - Sequential time ratio (*sequental portion*): $\alpha = \frac{s}{s+p}$
      - p: time required to execute the parallelizable portion.
        - Parallelizable time ratio (*parallel portion*): $1 - \alpha = \frac{p}{s+p}$
  - Speedup with n workers (**Amdahl's law**):

$$\sigma_n = \frac{T_1(P)}{T_n(P)} = \frac{s+p}{s+\frac{p}{n}} = \frac{1}{\frac{s+\frac{p}{n}}{s+p}} = \frac{1}{\frac{s}{s+p}+\frac{1}{n}\cdot\frac{p}{s+p}} = \frac{1}{\alpha+\frac{1}{n}\cdot(1-\alpha)}$$

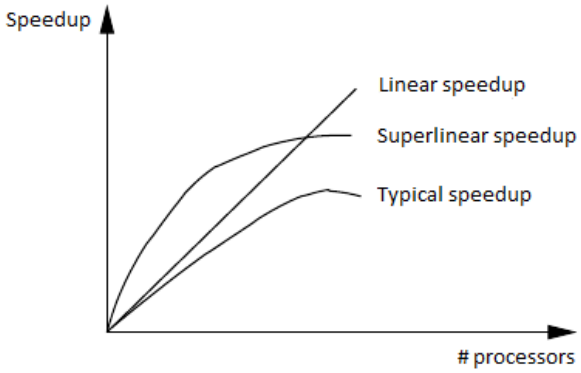
- σ_n, n relation. The parallel portion is $1 - \alpha$

  This is the **superlinear** (theoretical) speedup:



- When $n \to +\infty \Longrightarrow \begin{cases} T_n(P) = s + p/n \to s \\ \sigma_n = \frac{T_1(P)}{T_n(P)} = \frac{s+p}{s+\frac{p}{n}} \to \frac{s+p}{s} = \frac{1}{\alpha} \end{cases}$
  - $\alpha \to 0\ (p \to +\infty) \Longrightarrow$ **linear** speedup

- Real-life scenarios
  - Speedup will decrease after a certain point, because of:
    - Overhead due to context switches
    - Overhead due to communication among workers



  - Superlinear speedups can still happen thanks to **shared caches in multiprocessors**.
    - No overhead due to communication

o Gustafson's law
- The only difference is in the p time: this makes everything much more simple.
- $T_n(P) = s + p$
  - s: time required to execute the sequential portion.
    - Sequential time ratio: $\alpha' = \frac{s}{s+p}$

- p: time required by n workers to execute the parallelizable portion.
  - So: $T_1(P) = s + p \cdot n$
  - Parallelizable time ratio: $1 - \alpha' = \frac{p}{s+p}$
- **Speedup:** $\sigma_n(\alpha') = \frac{T_1(P)}{T_n(P)} = \frac{s+n \cdot p}{s+p} = \alpha' + n \cdot (1 - \alpha')$

Useful design patterns

Thursday, July 20, 2017
15:41

Factory pattern

Instead of invoking a constructor, the object creations is delegated to a *Factory* object.

```
// Starting objects

interface MyInterface {
    // Interfaces provide information hiding, the created
    // object can be of any type (that implements this interface)

// ...

}

class MyObject implements MyInterface {

// ...

}
```

```
// Factory

class MyFactory {

// ...


    public static MyInterface createMyObject() {

// ...

        return x; // x instanceof MyObject
    }
}
```

```
// Usage
```

```
MyInterface a = MyFactory.createMyObject();
```

- ## Singleton pattern

Special class for which only up to one instance at a time can exist.

- Possible thanks to **private constructors**
- With one just class: two possible initializations

```
class MySingleton {
    // Eager initialization
    private static MySingleton x = new MySingleton();

    public static MySingleton getSingleton(){
        return x;
    }

    private MySingleton(){ ... } // private constructor
}
```

```
class MySingleton {
    private static MySingleton x;

    // Lazy initialization
    public static MySingleton getSingleton(){
        if(x == null)
            x = new MySingleton();
        return x;
    }
    private MySingleton(){ ... } // private constructor
}
```

- With another class that provides the instance of the singleton
  - Static provider

```
class MySingletonPlace {
    // Same MySingleton instance for all MySingletonPlace objects
    private static MySingleton x;

    // Lazy initialization (also eager initialization is possible)
    public static MySingleton getSingleton(){
        if(x == null)
            x = new MySingleton();
        return x;
    }

}
```

  - Non-static provider

```
class MySingletonPlace {
    // MySingleton is unique for each instance of MySingletonPlace
    private MySingleton x;

    public MySingleton getSingleton(){
        if(x == null)
            x = new MySingleton();
        return x;
    }
}
```

Mutual exclusion

Wednesday, July 19, 2017
19:31

Shared-memory systems

- Multiple workers carry out some activities interacting with each other by using a common memory
- Cache coherence protocols are needed

Mutual exclusion

- When multiple workers try to modify the same memory location, a **race condition** occurs
- **Critical section**: the portions of code in which a shared resource is accessed are executed in **mutual exclusion** by processors.
    - In a single-processor environment, can be guaranteed by disabling interrupts, but it's inefficient for long critical sections.
- Mutual exclusion properties
    - **Safety**: only one process at the time can access the critical section
    - **Liveness**: eventually, all the processes requiring access to a critical section, will obtain it.
      It prevents:
        - Deadlock
        - Starvation
    - **Fairness**: processes acquire the lock in the same order they required it.
      It prevents starvation.

volatile

Friday, July 21, 2017
17:22

What it means

"Variable which might be accessed by other processes".

What does it do

- **It flushes everyone's cache when the variable is modified**, so all the other parties will see the freshest version of the data.
- **Atomic reads/writes** (not increments and swaps) on all primitive variables.

- Reads and writes are already atomic on all primitive variables, except for lond and double types: in order for them to have atomic reads and writes, they will need the `volatile` keyword.
  - This means that changes to a `volatile` variable are always visible to other threads.
- Increments and swaps are not atomic, even whit the `volatile` keyword:

```
volatile int i = 0;

void incBy5() {
    i += 5;
}
```

If two threads read the the `i` variable at the same time, the final result will be `5` instead of `10`.

Flags and turns

Thursday, July 20, 2017
12:13

Solutions to the mutual exclusion problem

Solution 1: boolean variable

```
volatile boolean free = true;

// prologue
while(!free);
free = false;

// critical section

// epilogue
free = true;
```

`volatile` keyword

"Variable which might be accessed by other processes".
It tells the compiler not to perform optimizations (`while(false)`) on portions of code containing that variable.

Problem: mutual exclusion not guaranteed

Two processors may read the `free` variable at the same time.
This variable should be read and written in an uninterruptible fashion.

Solution 2: flags

- A flag for each process

```
volatile boolean flag0 = false; // true if P0 in critical section
volatile boolean flag1 = false;
```

```
// Process P0
flag0 = true;
while(flag1);

// critical section

flag0 = false;
```

```
// Process P1
flag1 = true;
while(flag0);

// critical section

flag1 = false;
```

Problem: deadlock

If both processors set their flag subsequently.

Solution 3: turns

- Each process waits for its turn to access the critical section
- Once finished, it will pass the turn to the next process

```
volatile int turn = 0; // when = 0, it's P0's turn
```

```
// Process P0
while(turn != 0);

// critical section

turn = 1;
```

```
// Process P1
while(turn != 1);

// critical section

turn = 0;
```

Problem: always the same order

A process can't access the critical section twice subsequently.

Solution 4: Peterson's solution

- Correct solution with flags and turns

```
volatile int turn = 0;          // when = 0, it's P0's turn
volatile boolean flag0 = false; // true if P0 in critical section
volatile boolean flag1 = false;
```

```
// Process P0
flag0 = true;
turn = 1; // says that it's P1's turn
while(flag1 && turn == 1);

// critical section

flag0 = false;
```

```
// Process P1
flag1 = true;
turn = 0; // says that it's P0's turn
while(flag0 && turn == 0);

// critical section

flag1 = false;
```

- The last process that modifies the `turn` variable spins.

Correctness proof by contradiction

- Contradiction: both P0 and P1 are inside the critical section
- Both `flag0` and `flag1` are set `true` durong the pologue
- Suppose P0 entered first: `turn` must be `0`, otherwise P0 wouldn't have passed the `while()` check
    - If P1 has entered the critical section as well, `turn` must be 1, otherwise P1 wouldn't have passed the `while()` check ⟹ **contradiction**

Locks

Thursday, July 20, 2017

Target

```
public interface Lock {
    void lock();
    void unlock();

    // Other methods
    Condition newCondition();
    boolean tryLock();
    boolean tryLock(long time, TimeUnit unit);
    void lockInterruptibly();
}

public class MyLock implements Lock {
    // Actual lock/unlock implementations
    @Override
    public void lock() throws … {…};

    @Override
    public void unlock() {…};

    // If no clue about the others
    @Override
    public Condition newCondition() {
        throw new java.lang.UnsupportedOperationException();
    }

    // etc...
}
```

```
Lock l = new MyLock();

try {
    l.lock(); // prologue

    // critical section
} finally { // exceptions won't deny the unlock
    l.unlock(); // epilogue
}
```

Bakery's algorithm

- Inspired by ticket-based queues
  - Process A's ticket is N_a
- Works with multiple threads
- Each process has an unique ID (pid_a)
  - In case two processes acquire the same ticket.
- Steps
  - Process fetches a ticket (max ticket number + 1)
    - It can happen that two processes acquire the same ticket.
      In this case, the process with the lower ID wins.
    - Complete condition
      $(N_a < N_b) \parallel ((N_a == N_b) \&\& (pid_a < pid_b))$
- Code

<<Bakery.java>>

```
import java.util.ArrayList;
import java.util.List;

public class Bakery {
    public static final int NUM_THREADS = 10;

    // Contains tickets a (numbers) for all threads for a shared
        // 0: "not trying to enter" (default values)
        // value: arriving position (the higher, the later it arr
    List<Integer> ticket = new ArrayList<>(NUM_THREADS);

    // true when a thread is entering the queue, i.e. fetching a
    List<Boolean> entering = new ArrayList<>(NUM_THREADS);

    public void lock(int pid) { // pid: thread id
        // Entering the queue and fetching the ticket, set enteri
        entering.set(pid, true);

        // Searching for the max ticket
        int max = 0;
        for(int i = 0; i < NUM_THREADS; i++){
            int current = ticket.get(i);
            if(current > max)
                max = current;
        }
        ticket.set(pid, max + 1); // acquiring (max + 1)

        // Queue entered and ticket fetched, reset entering to fa
        entering.set(pid, false);
```

```
        // Check if it's someone else's turn
        for(int i = 0; i < NUM_THREADS; i++){
            if(i != pid){
                while(entering.get(i)){
                    /*  If thread i has the entering variable to
                        it is possible that it was accessing the
                        list at the same time as this thread (pid
                        thread i hasn't finished taking the ticket
                        Since thread i and this thread might have
                        same ticket at the end, this thread waits
                        thread i finishes taking its ticket. */
                }
                while   // If it's someone's else turn
                        (ticket.get(i) != 0 &&
                            // If the other thread i has higher p
                            (ticket.get(pid) > ticket.get(i) ||
                                // Same ticket case. Lower PID wi
                                (ticket.get(pid) == ticket.get(i)
                            )
                        ) {
                    // Thread pid waits until thread i finishes
                }
            }
        }
    }

    public void unlock(int pid){
        ticket.set(pid, 0);
    }
}
```

- Lower ticket wins
- Same ticket? Lower PID wins

Hardware solutions

Thursday, July 20, 2017
14:36

- *Testing* and *acquiring* actions must be perfomed **atomically**

• Test and set

```
int test_and_set(int* p, int new_val) {
    int old_val = *p;
    *p = new_val;
    return old_val;
}
```

```
void lock(int* p) {
    // Writes 1 in the lock in any case
    while(test_and_set(p, 1) == 1);
}
```

```
void unlock(int* p) {
    // Reading is useless
    *p = 0;
}
```

- **Not fair**: the acquisition order may be random (if more than one process tries to acquire the lock simultaneously).
  With a random acquisition order, **starvation** could still happen, but with a negligible probability.

Compare and swap

- Writes `new_val` only if `old_val == exp_val` (passed as parameter)

```
int compare_and_swap(int* p, int exp_val, int new_val) {
    int old_val = *p;
    if(old_val == exp_val) // writes conditionally
        *p = new_val;
    return old_val;
}
```

```
void lock(int* p) {
    // Sets the lock to 1 only if it was equal to 0
    while(compare_and_swap(p, 0, 1) == 1);
}
```

```
void unlock(int* p) {
    *p = 0;
}
```

- Same problem as the `test_and_set`

Fetch and add

- Atomic increment

```
int fetch_and_add(int* p) {
    int old_val = *p;
    *p = old_val + 1;
    return old_val;
}
```

- Can be used to implement a ticket-based locking solution, similar to the Bakery algorithm

```
record lock_t {
    int ticket_number; // next free ticket (for queueing)
    int turn;          // who has to be served
}

void lock_init(lock_t* l) {
    l->ticket_number = 0;
    l->turn = 0;
}
```

```
void lock(lock_t* l) {
     int my_turn = fetch_and_add(&(l->ticket_number));
     while(l->turn != my_turn);
}

void unlock(lock_t* l) {
     fetch_and_add(&(l->turn));
}
```

- Fair: processes acquire the lock in the same order they required it.

Spinning and sleeping locks

Thursday, July 20, 2017
15:26

General lock acquisition scheme

```
while(!acquire(lock)) {
     <waiting algorithm>
}

// critical section

release(lock);
```

- The waiting algorithm defines the lock type
  - **Spinning** lock: no operation
    - **Busy waiting** wastes CPU cycles
    - Good for short critical sections
  - **Sleeping** lock: `yield()`, changes the process state from *running* to *ready*
    - They involve **context switches**
    - Acquiring latency
    - FIFO queues containing blocked processes' PIDs to provide fairness
  - **2-phases lock**: hybrid between spinning and sleeping locks
    - At first, process spins
    - After a timeout, process suspends itself
    - **Barging**: when fairness is not mandatory, it's better to wake up spinning processes rather than sleeping ones.

Condition variables

Friday, July 21, 2017
14:17

- Objects that represent conditions on which processes may block themselves, waiting from some event to occur
  - Logical conditions. If true, the process is allowed to continue its execution
- `Condition` interface
  - Associated with locks, thanks to the `Lock::newCondition()` method
    - A thread must holding the corresponding lock, before waiting on a condition
  - Methods
    - `await()`: the invoker blocks on that condition
    - `notify()`: unblocks **a** thread blocked on that condition
    - `notifyAll()`: unblocks **all** threads block on that condition

Producer-Consumer problem

- With just **one variable** (buffer has only one data location)
- Look at the Java documentation guide

Javadoc's example

- Conditions provide a means for one thread to wait until notified by another thread **that some state condition may now be true**.
  The notifying thread knows when the codition becomes true or not, so that's why **condition variable names describe the condition state once it's possible to notify**.

```
class BoundedBuffer {
     final Lock lock = new ReentrantLock();
     final Condition notFull  = lock.newCondition();
     final Condition notEmpty = lock.newCondition();

     final Object[] items = new Object[100];
     int putptr, takeptr, count;
```

```
public void put(Object x)
     throws InterruptedException
{
     lock.lock();

     try {
          while(count ==
          items.length)
               // waits until it's
               notFull again
               notFull.await();

          items[putptr] = x;
          if(++putptr ==
          items.length)
```

```
public Object take()
     throws InterruptedException
{
     lock.lock();

     try {
          while(count == 0)
               // waits until it's
               notEmpty again
               notEmpty.await();

          Object x = items[takeptr];
          if(++takeptr ==
          items.length)
               takeptr = 0;
```

```
            putptr = 0;
        ++count;

        // indicating that buffer
        is "notEmpty"
        notEmpty.signal();
    } finally {
        lock.unlock();
    }
}
```

```
        --count;

        // indicating that buffer
        is "notFull"
        notFull.signal();
        return x;
    } finally {
        lock.unlock();
    }
}
```

Bounded buffer example

- Producer-Consumer problem with multiple variables (bounded buffer)
  - CDS_02_BOUNDEDBUFFER (Lab2)

Synchronized vs Condition variables

	Synchronized	**Condition variables**
Who's waked up	All threads	Only possibly interested threads
Fairness	No fairness policy can be declared	The ReentrantLock() constructor accepts an optional fairness parameter: if set to true, the lock favors granting access to the longest-waiting thread.

Counting semaphores

Friday, July 21, 2017
17:50

What are they

- Lock generalization
- Difference:
  - A lock allows up to one process to enter a critical section
  - A counting semaphores allows N concurrent accesses

In Java

- Class Semaphore
- The constructor requires two parameters:
  - Number N (*permits*) of threads that can access the critical section concurrently
  - Fairness (FIFO granting of permits)
- `acquire()`
- `release()`

Bounded buffer example

- Producer-Consumer problem with multiple variables (bounded buffer)
  - CDS_02_BOUNDEDBUFFER (Lab2)
- Code

```java
public class SemBoundedBuffer<T> implements BoundedBufferInterface<T> {
    // Semaphores: see constructor
    private final Semaphore availableItems, availableSpaces;

    // Buffer variables
    private final T[] buffer;
    private int putIndex = 0, takeIndex = 0;

    public SemBoundedBuffer(int capacity) {
        if (capacity <= 0) {
            throw new IllegalArgumentException();
        }
        availableItems = new Semaphore(0);
        availableSpaces = new Semaphore(capacity);
        buffer = (T[]) new Object[capacity]; // just a trick
    }

    @Override
    public void put(T x) throws InterruptedException {
        availableSpaces.acquire();

        int i = putIndex;
        buffer[i] = x;
        putIndex = (++i == buffer.length) ? 0 : i;

        availableItems.release();
    }

    @Override
    public T take() throws InterruptedException {
        T item;

        availableItems.acquire();

        int i = takeIndex;
        item = buffer[i];
        buffer[i] = null;
        takeIndex = (++i == buffer.length) ? 0 : i;

        availableSpaces.release();

        return item;
    }
}
```

Mutual exclusion in Java

Packages
`java.util.concurrent`
`java.util.concurrent.locks`
`java.util.concurrent.atomic`

Lock interface
Used to define some custom locking policy.

Pre-existing locking solutions
- Do not provide fairness by default: they use barging
  - Still possible to avoid barging thanks to some arguments in the constructor
- `tryLock()`: a thread tries to acquire a lock until a timer fires
- `isLocked()`

- `ReentrantLock` class
  - Solves the problem of a thread that tries to acquire a lock it already has.
    The locking operation just does nothing.
- Amotic standard data types
  - CDS_01_DATARACE (Lab1, explained here)
  - They allow atomic read and write operations
  - `AtomicInteger` class
    - `getAndSet()`
    - `compareAndSet()`
      Returns:
      - `true` if the old value was equal to the expected value
      - `false` if the old value was different to the expected value (so it won't write the new value)
    - `getAndAdd()`
    - …

Lab1 explained
Friday, July 21, 2017
11:29

- SharedCounter: interface representing how a counter should be implemented
  - The actual counter class (static nested classes inside the program class) changes in every program
- ThreadSeqID: class that gives threads a sequential ID, just for convenience

- Competitor: thread that tries to get the counter value for 10 times.
  The actual counter class almost always implements the get() method as a getAndIncrement() value (that returns the old value).
  So basically, competitors increment the timer 10 times.

Examples

- CdsDataRace01: this implements a counter (as a static nested class) with just an integer. This solution is not thread-safe because the read and write operations on the counter are not performed atomically.
  - CdsDataRace01Fixed: this implements a counter that uses the AtomicInteger class, so it works.
- CdsDataRace02: this implements a counter whose variable has the ThreadLocal<Integer> type. This means that **each thread sees its own variables**, so at the end every thread counts from 1 to 10 (displaying from 0 to 9, precedent values) just as they had their own private variables.
- CdsDataRace03: two threads (just two because they use the Peterson's lock solution) try to modify a map data structure.
  - The competitor this time has a different run method, that's why another class is used: CompetitorMap
    - They just insert a random integer from 0 to MAXELEMS (value = key in this example) if the element indexed by key is empty; otherwise, they remove it.
  - In this example, the Peterson's lock solution is implemented correctly so everything works fine.

Monitors

Friday, July 21, 2017
18:41

- Lock + condition

1. Hoare's monitor

- Components
  - Lock that protects critical section
  - 1 blocking **enter** queue
    - A process/thread blocks itself when the initial lock is occupied
  - More blocking **condition** queues
    - A process/thread blocks itself when a particular condition is not satisfied
    - A process/thread before blocking itself on a conditions, it frees the lock ("exits the monitor"), allowing other processes/threads to enter
- Once a thread inside the monitor invokes a signal()
  - It can wait on the enter queue and let the new thread come in (**signal and wait**)
    - It can wait on an *urgent* queue and let the new thread come in (**signal and *urgent* wait)**.
      This urgent queue has higher priority than the normal enter queue
  - It can leave the monitor and let the new thread come in (**signal and return**)

2. Mesa style monitor

- Once a thread inside the monitor invokes a signal()
  - It continues its execution while the freed process is moved to the enter queue (**signal and continue**)
  - This avoids context switches
  - The condition may change again while the freed process waits in the enter queue, so the contidion must be tested again before entering

Java monitors

Tuesday, September 12, 2017
10:39

- Mesa style
  - Once a thread inside the monitor invokes a signal() it continues its execution while the freed process is moved to the enter queue (**signal and continue**)
- There's only **one condition queue**, regardless of the actual number of logical conditions
  - `notifyAll()` method
  - Waking up all processes creates a lot of overhead
    - Explicit locks + condition variables are more efficient
- Each `Object` has a monitor associated to it
  - Therefore each `Object` provides methods like `wait()`, `notify()` and `notifyAll()`
  - `synchronized` refers to an object **istance**

```
public synchronized void method() {
    // ...
}
```

```
public void method() {
    synchronized(this) {
        // ...
    }
}
```

  - It can be done also for **classes**

```
synchronized(x.class) {
    // ...
}
```

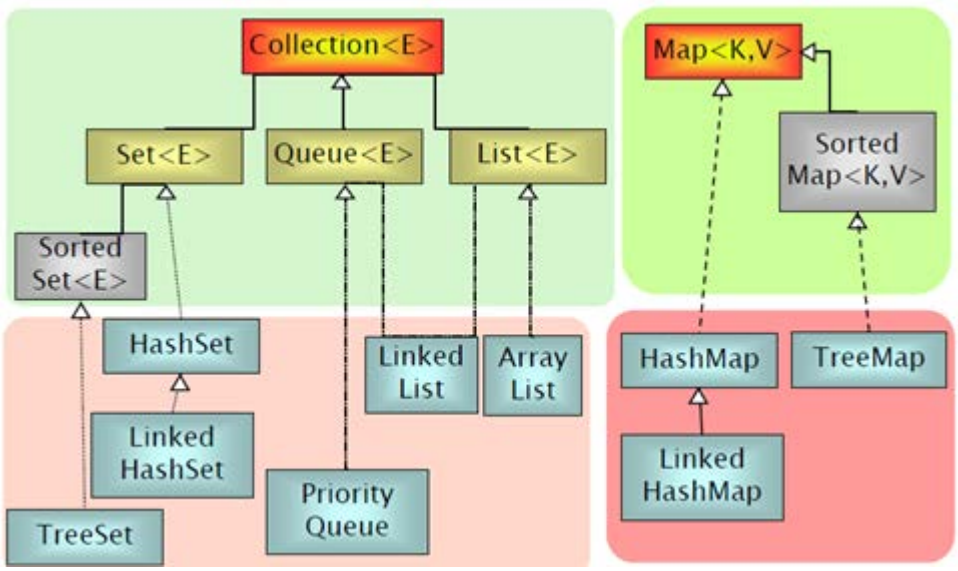
    - Java **Reflection paradigm**: each class is associated to a `Class` object

Collections

Saturday, September 23, 2017
18:12

Java Collection framework

- Set of classes that implement in efficient way common data structures
- Also called **Containers**
  - Group containers: group of objects
  - Associative containers: <key, value> pairs



- Set<E>: no duplicates
- Queue<E>: duplicates; sorted according to some rules
  - `compareTo(Object o)`
  - A ClassCastException might be generated
- List<E>: duplicates; no order

Iterating over a Collection

- Standard while/for loop
- foreach
  - Associative containers (e.g.: Map) should be converted into a group container with this `for(MyElement me : myMap.values()) { ... }`
- Iterators
  - `Iterator` is an interface
  - Example:

```
Iterator<MyElement> i = myCollection.iterator();
while(i.hasNext()) {
     MyElement me = i.next();
     // ...
}
```

  - It's unsafe to modify a collection while iterating over it (`ConcurrentModificationException`) unless appropriate methods are used (`add()` and `remove()`)

Thread-safe collections

Saturday, September 23, 2017
19:30

- Collections are **not thread-safe**

Reader-Writers problem (Java's `ReadWriteLock`)

- Shared resource
- Concurrent reads are allowed

- During a write, no other operations are allowed
- Lock for both reading/writing: not optimal
- Simple lock solution
  - Structure
    - Lock l: protects the two following variables
      - int r: number of processes currently reading
      - boolean w: true when a process is writing
    - Condition c
  - Reading

```
l.lock();
while(w == true)
     c.await();
++r;
l.unlock();

// reading

l.lock();
--r;
if(r == 0)
     c.notify();
l.unlock();
```

  - Writing

```
l.lock();
while(w == true || r > 0)
     c.await();
w = true;
l.unlock();

// writing

l.lock();
w = false;
c.notify();
l.unlock();
```

  - Problem: lots of readers could cause writers starvation
- Java's `ReadWriteLock` interface
  - readLock() returns a Lock for reading
    - This lock can be held by more processes at the time
  - writeLock() returns a Lock for writing
    - This lock is exclusive
  - Usage

```
rwl.readLock().lock();

// reading

rwl.readLock().unlock;
```

```
rwl.writeLock().lock();

// writing

rwl.readLock().unlock;
```

Executor

Sunday, September 24, 2017
17:12

- Task
  - What is done by threads
    - `Thread` constructor argument
  - Can implement the `Runnable` Interface
    - `run()` method
    - No return
  - Can be a `Callable` object
    - Allows to return a value
    - When this object is submitted to a thread pool, a `Future<T>` object is returned and it's used to check the state of the execution
- **Thread pool**
  - Threads fetch tasks from a **Job queue**

Frameworks

- `Executor`
  - Provides an **interface** through which a task can be executed with `execute(Runnable task)`
    - The task execution is **delegated to the framework**
    - Just `Runnable` objects
- `ExecutorService`
  - Executor extension
  - Both `Runnable` and `Callable` objects
  - Additional methods
    - `submit()` returns a `Future` object, to check the execution state
    - `shutdown()` waits for a task currently in execution to be completed before shutting down (it could take time)
    - `shutdownNow()` kills possible running tasks
  - Instantiated with `Executors.newFixedThreadPool(int n)`, creating a thread pool
    - Thread pool size
      - Too many will waste lot of resources and will cause too many context switches
      - Optimal number: #cores/CPUs
- `ScheduledExecutorService`
  - Allows
    - Delayed threads
    - Periodical threads
  - Instantiated with `Executors.newScheduledThreadPool(int n)`, creating a thread pool

- Methods
  - `schedule(Runnable/Callable task, long delay, …)`
  - `scheduleAtFixedRate(`
    `Runnable task,`
    `long initialDelay,`
    `long period, …`
    `)`
    - Next execution starts after `period` ms after the **start** of the current task
  - `scheduleWithConstantDelay(`
    `Runnable task,`
    `long initialDelay,`
    `long delay, …`
    `)`
    - Next execution starts after `delay` ms after the **end** of the current task

Double-check locking

Sunday, September 24, 2017
16:27

- Classing Singletons don't work in multithreaded environments because 2 thread could reach the `if` statement concurrently and create two instances

```
class MySingletonPlace {
     private MySingleton x;

     public MySingleton getSingleton() {
          if(x == null)
               x = new MySingleton();
          return x;
     }
}
```

- First solution: x variable protected in mutual exclusion with a `synchronized` block

```
class MySingletonPlace {
     private MySingleton x;

     public MySingleton getSingleton() {
          synchronized(this) {
               if(x == null)
                    x = new MySingleton();
          }

          return x;
}
```

  - Not optimal: threads will have to acquire the lock even just to check that `x != null`
- Second solution: threads won't have to acquire the lock if the instance already exists

```
class MySingletonPlace {
     private MySingleton x;
```

```
    public MySingleton getSingleton() {
        if(x == null) { // First check: avoids acquiring a lock if the instance already
        exists
            synchronized(this) {
                if(x == null) // Double check: w/o this, still 2 instances could be
                created
                    x = new MySingleton();
            }
        }

        return x;
}
```

- Another thread may acquire the instance before its construction ends, obtaining a reference to an object in an inconsistent state
- This could be solved declaring the Singleton `volatile`
- It's not efficient because the JVM avoids performing optimizations
- A non-`volatile` temprary support variable could solve this proble

```
class MySingletonPlace {
    private volatile MySingleton x;

    public MySingleton getSingleton() {
        MySingleton temp = x;

        if(temp == null) { // check on temp 'cause it's more performant
            synchronized(this) {
                temp = x; // someone could have updated x
                if(temp == null)
                    x = temp = new MySingleton();
            }
        }

        return temp;
}
```

- Most of the time, `temp` is already initialized
- The `volatile` variable `x` is accessed only once (first assignment) because of the "`return temp;`" instead of "`return x;`"

Nonblocking algorithms

Sunday, September 24, 2017
18:51

- Alternative to lock-based algorithms
- Using no locks
  - Improves concurrency
  - Avoids deadlocks

CAS and nonblocking counter

Sunday, September 24, 2017
18:54

- CAS

compareAndSet

```
public final boolean compareAndSet(V expect,
                                   V update)
```

Atomically sets the value to the given updated value if the current value == the expected value.

Parameters:

`expect` - the expected value

`update` - the new value

Returns:

true if successful. False return indicates that the actual value was not equal to the expected value.

- Solution provided by the `JavaAtomicInteger` class

```
public class NBCounter {
     private AtomicInteger count;

     public int increment() {
          int v;

          do {
               v = count.get();
          }
          /* compareAndSet() returns false (and hence the
             loop continues) if v is not the current value
             in count. This happens when someone else has
             modified count. */
          while(!count.compareAndSet(v, v + 1));
     }

     // Getter
     public int getCount() { return count.get(); }
}
```

ABA problem

Monday, September 25, 2017
10:05

1. $T_1 \xleftarrow{reads} A$
2. $T_2 \xrightarrow{writes} B$
3. $T_2 \xrightarrow{writes} A$

4. $T_1 \xleftarrow{reads} A$

- T_1 thinks nothing changed: it can be a problem in **lock-free** algorithms. Examples:
  - An object removed and then insterted again in a queue should be pointed by the same pointer for optimization
  - T_1 could be waiting for A to become B, and it could remain blocked
- A `compareAndSet()` succeeds, but it should fail instead
- Solution: "modified" counter
  - Doesn't work when counter overflows, it should be large enough

Treiber's stack

Sunday, September 24, 2017
19:10

- Non blocking solution for **shared stacks**
- Stack elements: `Node<E>`
  - Every element has a pointer to the next one
  - The stack itself is a pointer to the first element (`top`)
    - Implemented by `AtomicReference<Node<E>>`, which is an Atomic Reference to something (to `Node<E>` objects)
- Code

```
public class TreiberStack<E> {
     AtomicReference<Node<E>> top = new AtomicReference();

     /* Private static data structure used for implementing the
        Node<E> list of elements that will form the stack */
     private static class Node<E> {
          public final E item;
          public Node<E> next;

          public Node(E item) { this.item = item; }
     }


     public void push(E item) { // pushing on top
          Node<E> newTop = new Node<>(item);
          Node<E> oldTop;

          do {
               oldTop = top.get();
               newTop.next = oldTop;
          }
          /* compareAndSet() returns false (and hence the
             loop continues) if oldTop is not the current value
             in top. This happens when someone else has
             modified top. */
          while(!top.compareAndSet(oldTop, newTop));
```

```
        /* In this way, top is updated only when no one
           else updates top in the meantime.

           Remember that compareAndSet() is atomic.  */
    }

    public E pop() { // popping from top
        Node<E> oldTop;
        Node<E> newTop;

        do {
            oldTop = top.get();

            if(oldTop == null)
                return null;

            newTop = oldTop.next;
        }
        while(!top.compareAndSet(oldTop, newTop));

        return oldTop.item;
    }
}
```

- It suffers from the ABA problem because this solution doesn't use References, hence a thread cannot distinguish a new identical top value from the old one

Michael/Scott queue

Sunday, September 24, 2017
19:32

- Non blocking solution for **shared queues**
- Queue elements: Node<E>
  - Queue has a head and a tail reference
- put() Code

```
public class MSQueue <E> {
    private final Node<E> dummy = new Node<>(null);

    // Head and tail
    private final AtomicReference<Node<E≫ head = new
    AtomicReference(dummy);
    private final AtomicReference<Node<E≫ tail = new
    AtomicReference(dummy);

    private static class Node <E> {
        public final E item;
        public AtomicReference<Node<E≫ next;

        public Node(E item) { this.item = item; }
```

```
    }

    public boolean put(E item) {
        Node<E> newNode = new Node<>(item);

        while(true) {
            Node<E> curTail = tail.get();
            Node<E> tailNext = curTail.next.get();

            if(curTail == tail.get()) { // someone changed the tail in
            the meantime
                if(tailNext != null){ // someone insterted something in
                the meantime
                    tail.compareAndSet(curTail, tailNext); // update
                    tail if not yet
                }
                else { // no one insterted something in the meantime
                    /* Try to insert the new node. If the operation
                       succeeds, update tail */
                    if(curTail.next.compareAndSet(null, newNode)) {
                        tail.compareAndSet(curTail, newNode);
                        return true;
                    }
                }
            }
        }
    }
}
```

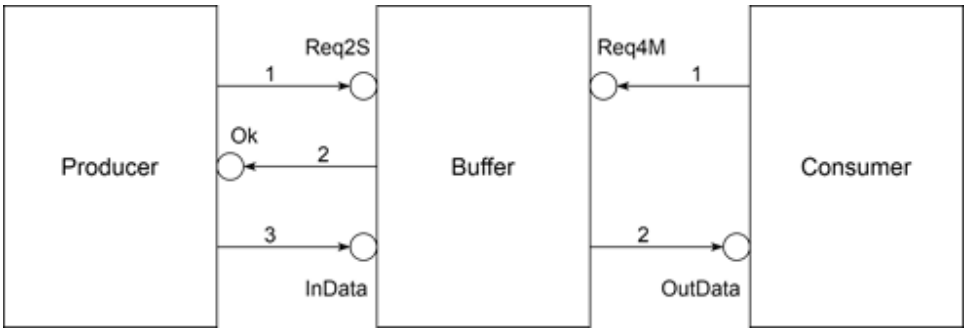
- The `AtomicStampedReference` is designed to be able to solve the A-B-A problem which is not possible to solve with an `AtomicReference` alone.

Message-passing systems

Monday, September 25, 2017
16:59

Logical ports

- Used in systems where memory is not shared
- Nodes use ports (logical ones) (many-to-one channel)
- Information is gathered in a `CommunicationEndpoint` class
    - Message **delivered** when it reaches local buffer
    - Message **received** when it's passed to the application (`CommunicationEndpoint` class)
    - Bounded-buffer protocol



  - Circles are `EndPoint` objects (logical ports)
  - The alternative to these logical ports would be to label every message
- **Synchronous** calls block the invoker, **asynchronous** calls let the invoker continue its execution
  - Send calls could be both
  - Receive calls are always synchronous
    - It could be **blocking** or **not-blocking** depending on the behaviour on reading an empty buffer

Channel

- Ideal: guarantees both ordering and reliability

Time and global states

Monday, September 25, 2017
10:28

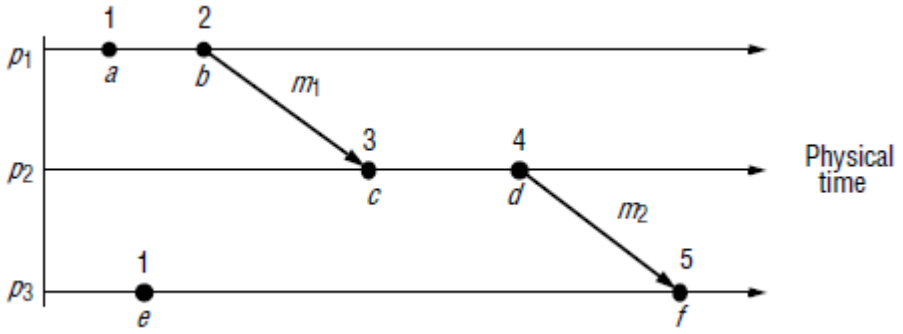
- It defines a unique **event order**
- Two solutions
  - Timestamps based on **logical clocks** instead of physical ones
  - Messages to sync clocks
- ***Local** happen-before* relation: $a \rightarrow_p b$ (p saw a happening before b)
- ***Global** happen-before* relation: $a \rightarrow b$ (everyone saw a happening before b)
  - Local synchronization
    $a \rightarrow_p b, \forall p \Longrightarrow a \rightarrow b$
  - Asynchronous communication $\Longrightarrow send(m) \rightarrow recv(m)$
  - Synchronous communicatiom $\Longrightarrow ssend(m) \rightarrow srecv(m) \Longrightarrow$
  - $\begin{cases} \forall h: srecv(m) \rightarrow h \Longrightarrow ssend(m) \rightarrow h \\ \forall g: ssend(m) \rightarrow g \Longrightarrow srecv(m) \rightarrow g \end{cases}$
  - Transitivity: $a \rightarrow b, b \rightarrow c \Longrightarrow a \rightarrow c$
  - Concurrency: $a \parallel b$ concurrent when neither $a \rightarrow b$ nor $b \rightarrow a$

Lamport's timestamps

Monday, September 25, 2017
10:41

- Each node n has a counter L_n
- $L(e)$ = timestamp associated with event e
- Counter incrementing rules

- L_n is incremented before each event is issued **at** node n, L_n is used to timestamp the event
- Upon exchanging a message
  - Sender
    - $t = + + L_i$ **(logical clock)**
    - $i \overset{m,t}{\longrightarrow} j$
  - Receiver
    - $L_j = \max(t, L_j)$
    - $\mathrm{L}\left(j \overset{m,t}{\longleftarrow} i\right) = + + L_j$
- Example



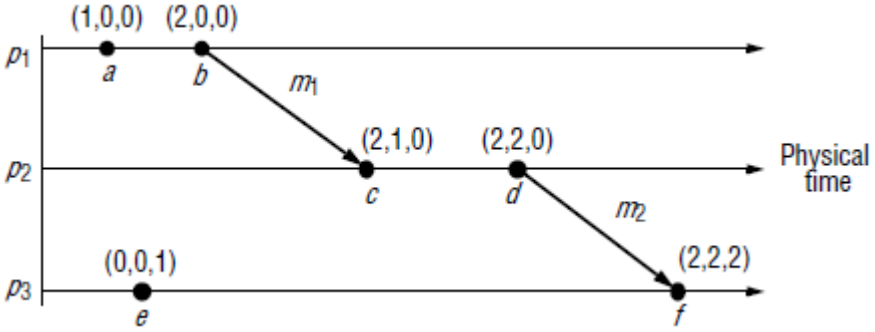
- Properties
  - $a \to b \Longrightarrow L(a) < L(b)$
  - $L(a) < L(b) \nRightarrow a \to b$
    - It could be that $a \to b$ or $a \parallel b$
    - If two timestamps are equal, the process with smaller ID is considered first

Vector timestapms

Monday, September 25, 2017
10:59

- They overcome the shortcoming Lamport's conclusion that $L(a) < L(b) \nRightarrow a \to b$
- 1 vector clock for N nodes
  - `int V[] = new int[N]; // initialized to 0, one for each node`
  - Piggybacked to messages
- Every i-th node has its own vector clock V_i
  - $V_i[i]$ is the number of events that i has timestamped
  - $V_i[j], j \neq i$ is the number of events occurred at node j that have **potentially affected** i
- $V(e)$: vector timestamp associated with event e
- Vector clock updating rules
  - $+ + V_i[i]$ everytime node i issues an event, then $V(e) = V_i$
  - Upon exhanging a message
    - Sender
      - $+ + V_i[i]$

        - $t = V_i$ **(logical clock)**
        - $i \overset{m,t}{\longrightarrow} j$
    - Receiver
        - $V_j[k] = \max\left(t[k], V_j[k]\right), \mathrm{k} = 1, \ldots, \mathrm{N}$
        - $++\mathrm{V_j}[j]$
        - $\mathrm{V}\left(j \overset{m,t}{\longleftarrow} i\right) = V_j$
    - Example



- Comparing vector timestamps
    - $V_i = V_j \Leftrightarrow V_i[k] = V_j[k]\ \forall k = 1, \ldots, N$
    - $V_i \leq V_j \Leftrightarrow V_i[k] \leq V_j[k]\ \forall k = 1, \ldots, N$
    - $V_i < V_j \Leftrightarrow V_i \leq V_j \wedge V_i \neq V_j$
- Properties
    - $a \rightarrow b \Longrightarrow V(a) < V(b)$
    - $V(a) < V(b) \Longrightarrow a \rightarrow b$
        - Determing the order by which events occured by looking at vector timestamps is called **casuality checking**
        - Proof by contradiction
            - $V(a) < V(b) \wedge a \nrightarrow b \Longrightarrow a \parallel b \vee b \rightarrow a$
                - If $b \rightarrow a \Longrightarrow V(b) < V(a)$, but it's not
                - If $a \parallel b$, it should be neither $V(a) \leq V(b)$ nor $V(b) \leq V(a)$
                    - If $V(a) \leq V(b)$ is not true, then not even $V(a) < V(b)$ is
                    - Contradiction
- Memory drawback: N integers of each vector, instead of just 1

Global state snapshots: Chandy-Lamport's algorithm

Monday, September 25, 2017
11:48

- Used to get a global state snapshot of a distributed system when the algorithm is executed
- Can be executed by everyone at anytime
- Snapshot
    - State of each process
    - State of each channel (set of messages)
- Each channel has to store its state + the state of its **incoming** channels
- Assumptions

- No failures
- Unidirectional channels
- There's a channel b/w any 2 processes (strongly connected graph)
- Any process can start the algorithm
- The algorithm can run in background

- Algorithm
  - It uses special messages called **markers**
    - **State** of an incoming channel: set of messages flowing through the channel since the fist marker was received on the channel
  - The initiator
    - Stores its state
    - Sends a marker over each outgoing channel
    - Starts recording incoming messages
  - Upon receiving a marker from channel c, a process
    - If state not stored yet
      1. Stores its state
      2. Stores the state of c as an empty set
      3. Sends a marker over each outgoing channel
      4. Starts recording incoming messages
    - else
      1. Stops recording messages arriving from c
      2. Stores the state of c as the set of recorder messages
  - Ending
    - Each process, after receiving n markers, sends the stored states to the initiator

Request-reply protocols

Saturday, July 22, 2017
15:13

Communication paradigms

- Direct communication
  - Sender knows the identity of receivers and vice-versa
  - Types

Req/Rep Request-reply	**RPC** Remote Procedure Call	**RMI** Remote Method Invocation
		• Object-oriented version of RPC

    - These mechanisms are built on top of the send/receive primitives
    - Higher level of abstraction for communication
- Indirect communication
  - Sender doesn't know the identity of the receiver.
  - There's some **middleware** in between.
    - *Message Oriented Middleware* (MOM)
    - Receivers **fetch** messages from the middleware
  - Parties communicating indirectly are called **loosely coupled**.

General functions

Client

- `doOperation()`
  - Calls a server's function
  - Waits (blocks) until server replies
  - Arguments
    - **Remote reference** (server's address)
    - Operation ID
    - [resource on which to perform the requested operatoin]
    - […]
  - Request message fields

`MessageType`	**0** (**request**) or 1 (reply). Needed because clients can act both and server and clients.
`RequestId`	Unique for each client
`RemoteReference`	Packet generator's address (in this case **the client's one**), useful to the server
`OperationId`	
`Arguments`	

  - **Marshalling**: the client needs to transform data to a standard format.
  - Performs an unmarshalling of the reply

Server

- `getRequest()`
  - Fetch a client's request
  - **Unmarshalls** the request message
  - Request response fields

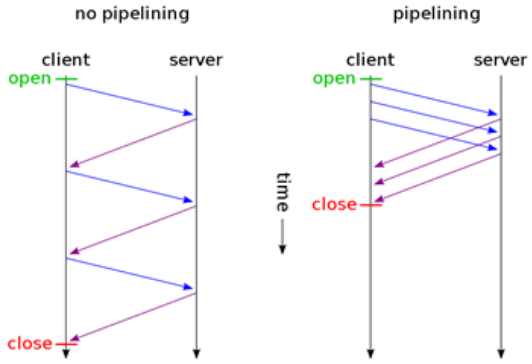
`MessageType`	0 (request) or **1** (**reply**). Needed because clients can act both and server and clients.
`RequestId`	Unique for each client
`RemoteReference`	Packet generator's address (in this case **the server's one**), useful to the server
`OperationId`	
`Arguments`	Contains the **result**

  - Performs the marshalling before sending the reply
- `sendReply()`
  - Aruments
    - Reply message to be sent
    - Client reference

Unrealiability

- Problems
  - Packets lost
  - Packets re-ordering
  - **Fail-stop failure**: processes may remain crashed foreve
- What can happen
  - The doOperation() method could keep waiting forever
    - Lost request message: **retransmission** if no reply is received after a timeout
      - Longer than 1 RTT + processing time
      - After a number of tries, it should stop: the server could have crashed
      - The server should **filter request message duplicates**, by looking at the `RequestId` and `RemoteReference` pair
    - Lost reply message: **result caching** in order for the server to reply to all duplicate requests
      - This is done to improve performance
      - Results may be cached both at the server and at another node (like proxies)
        The result may vary with the operation type:
        - Idempotent: same result if executed more than once (math, HTTP GET)

- Results can be cached
- The HTTP pipelining (palallel requests) is used for static content, hence for idempotent operations



- Non-idempotent: different results if executed more than once (increments, timestamps, HTTP POST)
  - Results cannot be cached

- Deleting cached results: the server can interpret each successive client request as an ACK for the previous reply, so it can delete previous cached results.
  It should delete all cached results after a limited period of time anyway (the client could have disconnected)
- The client should **filter reply message duplicates**, by looking at the `RequestId` and `RemoteReference` pair

RPC

Sunday, July 23, 2017
17:26

What does it stands for

Remote Procedure Call

Paradigm goal

- Extend the *procedure call* concept to distributed environments
- A client can call a prcedure and obtain a result as if it was calling it locally

Transparency

- Location transparency: the actual location where the computations are performed is hidden to the client
- Access transparency: local and remote procedures must be called in the same way
- Transparency with respect to the concurrency strategies applied to the server
- Transparency with respect to the communiction details between the client and the server
  - Send/receive primitives
- Transparency with respect to the actual send/receiver primitives implementations
- Transparency with respect to the underlying platform on which a node is built (OS, programming language, data un/marshelling, …)

Interfaces

- **Why**: clients must know which procedures are provided by the server.
  - This solution should be standard, even with heterogeneous clients.

- Clients only see interfaces, and not their actual implementation.
  - The interface should not be changed: implementations can be
- **IDL**: Interface Description Language
  - Platform-independent, can help solving heterogeneity problems
  - Examples: Sun XDR, CORBA IDL, WSDL
  - An IDL has a **compiler** which **translates the IDL code into interfaces** (in the proper language). This is transparent to the programmer.

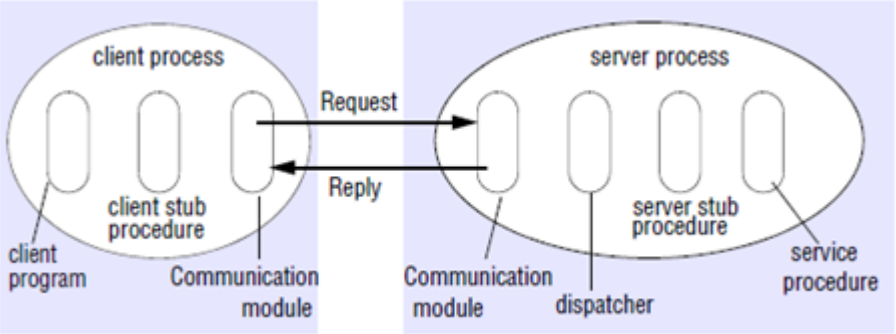
Semantics

- `in`, `out` and `inout` parameters. The latter are included both in the request and in the reply messages.
- RPC is built **on top of a request-reply protocol** that **could or could not be reliable** (request retransmission and duplicate filtering).
  Three types of **call semantics**:

Fault tolerance measures			*Call semantics*	
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*		
No	Not applicable	Not applicable	*Maybe*	
Yes	No	Re-execute procedure	*At-least-once*	← E.g. used by Sun RPC
Yes	Yes	Retransmit reply	*At-most-once*	← E.g. used by CORBA/Java RM (we will talk about RMI later)

  - Maybe: the procedure could be executed or not.
    When the underneath protocol is **not reliable**, the message could not arrive at destination.
    The local procedure should return after a timeout
  - At-least-once: even an error message is accepted. The operation could be performed more than once since retransmission without duplicate filtering is used.
  - At-most-once: the invoker knows that the result is executed exactly once.
    Both retransmission and duplicate filtering are used in this case.

PRC architecture



What needs to be written

- The client program and the service procedure
- Modern compilers do the rest of the job

- Client sends request message

Client process	Server process

• Stub procedure
  • Actual procedure invoked by the client in its program
  • Hides underlying details
  • Client thinks it's executed locally
  • Marshals arguments
  • Passed to the communication module
• Communication module
  • It sends the request message and receives the reply one
  • Deals with low-level communication details
  o Reliability
	• Communication module
	  • Low-level communication details
	  • It passes request messages to the dispatcher
	• Dispatcher
	  • Invokes server-side stub procedures after receiving request messages
	  • The request message is passed as an argument to that procedure
	• Stub procedure
	  • It unmarshals request message arguments
	  • Calls the service procedure
	• **Service procedure**
	  • Contains the actual procedure implementation, called with unmarshalled data by the stub procedure

Server replies

- **Server**
  - The service procedure returns the result
  - The stub procedure marshals the result, **inserting it into a reply message**
  - The communication module sends the reply
- **Client**
  - The communication module receives the reply message
  - The stub procedure unmarshalls the result, returning it to the client program

RMI

Thursday, August 10, 2017
11:24

In general

- Further development of RPC, introducing the **OOP** abstraction to distributed environments
- RPC was about remote procedures, RMI is about **remote objects**
  - The implementations still makes use of interfaces
  - Local and remote object must be placed in the **JVM heap**
    - The local object is called ***stub*** or ***proxy*** object.
      - Their methods act like *stub procedures* in RPC.
    - The target object is called the ***servant*** object.
      - The client can access the servant thanks to its ***remote reference***:

32 bit	32 bit	32 bit	32 bit	32 bit
IP address	**Port**	**Object creation time**	**Object #**	**Object interface**

      - The client only sees a reference to the proxy object
      - The remote reference is hidden

        - This association between the two objects is done by the *Remote Reference Module*
- RMI allows to **pass/return arguments** both by value and by reference
    - By value: an object is first serialized, sent in the request message and then de-serialized.
        - These objects must implement the `Serializable` interface.
    - By reference: thanks to *remote references*.
        - Clients don't see them, so they use local references instead:
          RMI retrieves transparently the remote one thanks to the *Remote Reference Module*.
            - Local references cannot be used directly bacuse they make no sense in remote locations.
        - An object cannot be passed by reference if no remote references are available for that object.
- RMI uses the ***At-most-once*** call semantic.

Initializing a remote object

- The interface

```
public interface RemoteInterface extends Remote { … }
```

    - Imported as a library both by the proxy and the servant.
    - Implemented by the servant object.
- The remote object

```
public class RemoteObject extends UnicastRemoteObject implements
RemoteInterface { … }
```

    - **Export operation**: makes the object available for remote access.
      A remote reference is then created.
        - It can also be done by calling `stub = UnicastRemoteObject.exportObject()`.
          This returns a local stub.
    - The remote object could also implement the Remote interface directly, but nobody does it.

RMI registry

- Naming service (like DNS) running on some node (whose address is known by everyone) that **converts symbolic names into remote references**.

Symbolic name	Remote reference
…	…
…	…

- Runs on port 1099 by default.
- How it can be launched
    - From command line
        - `rmiregistry` command
        - A reference to it can then be retrieved with

```
Registry myRegistry = LocateRegistry.getRegistry();
```

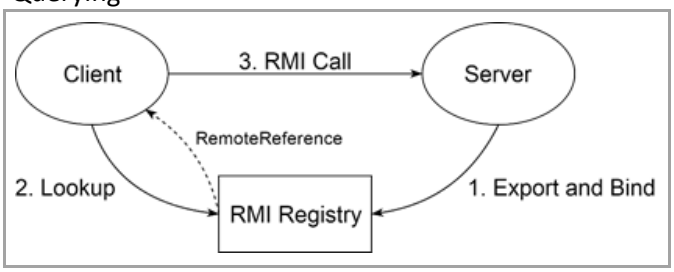
    - **From code**

```
Registry myRegistry = LocateRegistry.createRegistry(REGISTRY_PORT);
```

- ***Bindig***: registering a remote reference with a symbolic name.
    - **Symbolic names are known by the nodes of the system**, since they are defined by the programmer.

- Two ways
  - ```
    myRegistry.rebind(symbolic_name, stub);
    ```
    - `stub = UnicastRemoteObject.exportObject()` during the export operation.
  - **Using the `Naming` class** that follows the *Java Naming and Directory Interface (JNDI)*, which offers a set of static methods to interact with the RMI registry.
    ```
    Naming.rebind(registry_address + "/" + symbolic_name, servant);
    ```
    - *registry_address* = "//127.0.0.1:1099"
      - So a typical symbolic reference could be "//127.0.0.1:1099/Calculator"
    - It doesn't need a `stub`, so it doesn't require this call:
      ~~`stub = UnicastRemoteObject.exportObject()` during the export operation~~

- Querying

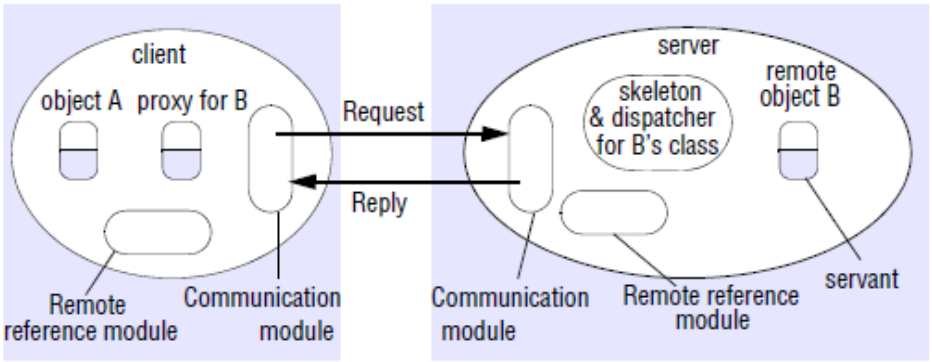


  - The **lookup** operation consists in a request message to the RMI registry, requesting a remote inference for a symbolic name.
    ```
    Registry myRegistry = LocateRegistry.getRegistry(registry_address);
    stub = myRegistry.lookup(symbolic_name);

    or

    stub = Naming.lookup(registry_address + "/" + symbolic_name);
    ```
    - It's the only non-transparent step in the client's code.
  - RMI calls can return other remote references and they can accept remote references as parameters (implicitly).
    - So usually the *RMI Registry* is only used at the beggining for a few objects: to obtain other remote references, RMI calls could be used instead.

RMI architecture



- Compilers write everything automatically
    ```

- Only the client program and the servant methods have to be written manually

Common parts

- Communication modules
  - Low-level communication details, like in RPC
  - The client one sends RMI requests to the server one, which forwards them to a dispatcher
  - The client one also receives reply messages from the server one
- *Remote Reference Module*s
  - Local/remote reference translation with the ***Remote Object Table***. It contains:
    - One entry for each servant object held by a node, containing **the local reference of the servant and the remote reference associated to that same servant**.
      - When an object is exported, a remote reference is created and stored in the table together with the corresponding local reference.
      - When a servant is bind at a Registry, the corresponding remote reference **is sent to the Registry** together with is symbolic name.
    - One entry for each proxy object held by a node, containing **the local reference of the proxy and the remote reference of the corresponding servant**.
      When a remote reference is returned for the first time by a lookup or an RMI call:
      - A proxy for the corresponding remote object is created and returned to the client
      - An entry containing the local reference of the proxy and the corresponding remote reference is inserted into the table.
  - How local and remote references are used
    - When a local reference is passed to a method, this *Remote Reference Module* retrieves the corresponding remote reference from the *Remote Object Table*.
    - When a remote reference is returned by an RMI call, this *Remote Reference Module* retrieves the corresponding local reference from the *Remote Object Table*.

Client	**Server**

• Proxy or stub
  • Local proxy object
  • It implements the same interface as the servant (same behaviour)
	• Servant
	  • Target object2
	• Dispatcher
	  • Receives request messages from the communication module
	  • Forwards them to the appropriate method in the skeleton
	• Skeleton
	  • Invoked by the dispatcher
	  • Unmarshals the arguments in the request message
	  • Invokes the corresponding method in the servant
	  • Marshals the result returned by the servant
	  • Translates local references into remote ones thanks to the *Remote Reference Module*
	  • Sends the result back to the client through the *Communication Module*

Multithread RMI remote invocations

- The programmer **only** has to guarantee the **thread-safety of data structures**

- Strategies
  - Thread per request
  - Thread per connection
  - Thread per object
- The strategy is not guaranteed:
  - *"A method dispatched by the RMI runtime to a remote object implementation may or may not execute in separate thread.*
    The RMI runtime makes no guarantees with respect to mapping remote object invocations to threads.
    Since remote method invocation on the same remote object may execute concurrently, a remote object implementation needs to make sure its implementation is thread-safe."

Class loading

- Non-remote objects are passed by values (`Serializable`) and remote ones are passed by reference as arguments and result of RMIs
  - If the recipient doesn't have a class of an object passed by value, **its code will be downloaded automatically**.
- Java *Class Loaders* load classes' code at runtime, to avoid big initial overheads.
  - They look for `.class` files and they decompress `.jar` file, if necessary, to load classes from them.
  - The are **multiple** *Class Loaders*, **hierarchically organized** (depending on locations and policies)
    - The Class Loader at the root starts searching for it: if it doesn't succeed, it **delegates** the job to the next Class Loader in the hierarchy.
    - If no class is found, a `ClassNotFoundException` is thrown.
  - When the JVM is started, three *Class Loaders* are used
    - Root: **Bootstrap** Class Loader: `<JAVA_HOME>/jre/lib` directory
    - **Extensions** Class Loader: `<JAVA_HOME>/jre/lib/ext` directory
    - **System** Class Loader: paths specified by the system property `java.class.path`.
      - By default is `.`, the current directory.
      - Can be changed with the `-cp` flag.
    - Other costum Class Loaders
      - Added at the bottom of the hierarchy
      - **They allow class downloading from remote locations**
      - *Network Class Loaders* or *Remote Class Loaders*.
        - E.g.: the [Java] Applet Class Loader downloads classes via HTTP.
        - RMI Class Loader: searches within the `java.rmi.server.codebase` system property
          - A `SecurityManager` **must** be set up before downloading code from remote locations

```
if(System.getSecurityManager() == null)
    System.setSecurityManager(new SecurityManager());
```

Garbage collection

- Only local objects: the JVM's GC periodically checks for objects with no more references and deletes them.
  - *Reference counting*: a counter keeps track of the number of references for each object.
    - This solution does not reveal reference dead-cycles.
  - *Mark and sweep*: objects are seen as graph nodes.
    - Marking: every object, starting from the root, is marked.

        - Sweeping: non-marked/non-reachable objects are destroyed.
- The distributed garbage collection is **based on *reference counting***.
- It works in cooperation with the local GC
    - The server's *Remote Reference Module* holds a data structure containing, for each servant object B, the set of client's IDs that are holding a reference to B. This set is called `B.holders`.
    - Adding/removing
        - When a client receives a remote reference to B, it also implicitly invokes the `addRef(B)` or `dirty(B)` method on the server; the latter will create and send a proxy for B and then it adds the client to `B.holders`.
        - When a client's GC is up to garbage collect a proxy object for B, it implicitly the `removeRef(B)` or `clean(B)` method on the server; the latter will then remove the client from `B.holders`.
    - When `B.holders` is empty, the server's GC will reclaim the space occupied by B unless there are any local holders.
    - Concurrency problem: one unique client could delete an object B while another client is requesting it.
      Instead of keeping `B.holders` empty, a **temporary dummy entry** is added until the `addRef()` of the other client arrives.
    - A client could crash before releasing an object B: a predifined timeout (**leasing solution**) is agreed b/w the client and the server.
      Clients are then forced to renew their leases before they expire.

Akka

Thursday, August 17, 2017
17:44

Based on the Actor Model

- Model applied to concurrent and distributed systems
- Working units are ***actors***
    - They can exchange messages among each other
    - They can run an handler once they receive a message
        - Messages can be created/forwarded
        - New actors can be created
        - An activity can be executed
    - Deployment
        - On the same machine (concurrency)
        - On different machines (distribution)
- A service is seen as many ***activities*** (processing, storageor communication), each one assigned to an *actor*.

The Akka framework

- <<Akka.pdf>>
- Actor can form hierarchies
    - The root is called the **Guardian System Actor**

Failure model

Friday, August 18, 2017
14:59

A process/channel that experiments failures is said to be **faulty**, otherwise it is said to be **correct**.

Failure types

Class of failure	*Affects*	*Description*
Fail-stop	Process	Process halts and remains halted. Other processes may detect this state.
Crash	Process	Process halts and remains halted. Other processes may not be able to detect this state.
Omission	Channel	A message inserted in an outgoing message buffer never arrives at the other end's incoming message buffer.
Send-omission	Process	A process completes a *send* operation but the message is not put in its outgoing message buffer.
Receive-omission	Process	A message is put in a process's incoming message buffer, but that process does not receive it.
Arbitrary (Byzantine)	Process or channel	Process/channel exhibits arbitrary behaviour: it may send/transmit arbitrary messages at arbitrary times or commit omissions; a process may stop or take an incorrect step.

Distributed algorithms

Friday, August 18, 2017
14:11

Typical problems

- Distributed mutual exclusion
- Leader election
- Reliable multicast
- Consensus
- Spanning tree

Comparison metrics

Traditional algorithms	**Distributed algorithms**
• Time complexity • Memory complexity	• Number of messages required (communication is very time consuming)

Distributed mutual exclusion

Friday, August 18, 2017
14:15

Requirements

- Safety: just one node into the critical section
- Liveness: no starvation; eventually all nodes must access the critical section

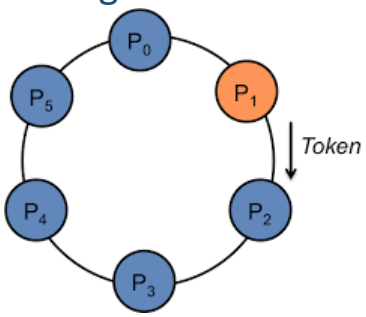
Techniques

- Token-based
- Non-token-based
- Voting schemes (quorum algorithms)

Centralized solution

- A *Manager* or *Coordinator* node grants access to the critical section.
  - Nodes send a `request`, the manager permits OK and nodes have to `Release`.
  - 3 messages $\Longrightarrow$ message complexity is $O(1)$
- Concurrent requests end up in a requests queue
- The manager is a performance bottleneck
- Not fault tolerant
  - The manager is a single point of failure
    - If nodes can detect failures, a new manager can be elected
  - If a node currently into the critical section crashes, the resource is never released
    - If nodes can detect failures, the manager can revoke the permit

Token-ring



- Message complexity is $O(N)$: worst-case scenario involeves just one node to request the critical section, but al the other $N-1$ nodes have to **forward** to token without using it.
- Fully distributed/decentralized
- Not fault tolerant
  - When a node crashes, the ring is interrupted
    - If nodes can detect failures, they can rebuild it
  - If a node currently owning the ticket crashes, the resource is never released
    - If nodes can detect failures, the new token can be regenerated by a designated node that has to be elected with an election algorithm

Ricart - Agrawala algorithm

- A node sends a request message to every other node and waits for a reply from all of them
  - Request messages contain timestamps, to define a total order among the events
- Upon receiving a request, a node:
  - If it doesn't want to access the critical section, it replies immediately

- If it is in the critical section, it records this pending request and it will reply as soon as it gets out
- If the node currently wants to access the critical section too, it compares timestamps. It replies if it loses, otherwise it will enter the critical section.
- Message complexity is $O(N)$ because $2(N-1)$ messages are sent in total ($N-1$ requests by the requesting node, $N-1$ responses from all the other nodes)
    - This algorithm allows different requests and replies to be sent in parallel, resulting in a message complexity of $O(1)$
- Crashing nodes won't reply the to requesting one
    - The requesting node could decrement `pendingReplies` whenever it detects a failure
- Pseudo code

```
boolean myActiveRequest = false;
int pendingReplies = 0;
TS reqTime; // timestamp of a potential request
ProcEndPoint myEp; // local communication endpoint
Collection<ProcEndPoint> peerProcs;
Collection<ProcEndPoint> toBeReplied;

void requestCS(){
    Message reqMsg = new Message(REQ); // this constr. sets reqMsg.da
    reqTime = currentTS();
    reqMsg.TS = reqTime; // piggyback current timestamp in the reques
    myActiveRequest = true;
    pendingReplies = N - 1; // N: number of nodes
    for(ProcEndPoint ep : peerProcs)
        ep.send(reqMsg);
}

void inMsgHandler(){ // callback method invoked upon message arrival
    Message inMsg;
    ProcEndPoint ep = myEp.recv(inMsg) // received message inMsg, sen
    switch(inMsg.data){
        case REQ:
            if(!myActiveRequest || reqTime.greaterThan(inMsg.TS))
                ep.send(new Message(OK));
            else toBeReplied.add(ep);
        case OK:
            pendingReplies--;
            if(pendingReplies == 0){
                <enter critical section>
                <release critical section>
            }
    }
}

void releaseCS(){
    myActiveRequest = false;
    Message okMsg = new Message(OK);
    for(ProcEndPoint ep : toBeReplied)
        ep.send(okMsg);

    toBeReplied.clear();
}
```

Leader election

Friday, August 18, 2017
14:42

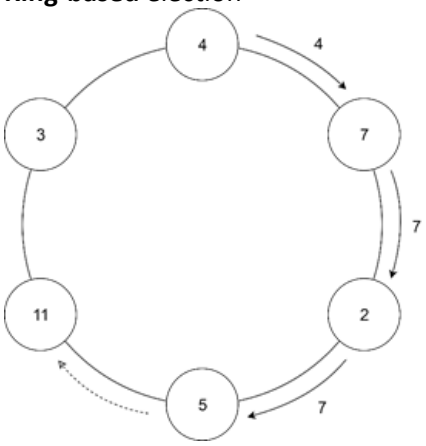
- Processes have an unique ID
- Must be fault tolerant
- Steps
    1. Find the leader: process with **highest ID**
    2. Inform all other nodes about it

(processes stores the leader ID, `null` if there's still no leader)

- Requirements
  - Safety: all correct processes must agree on the leader ID
  - Liveness: eventually, the election will end

Chang & Robert's algorithm

- **Ring-based** election



- Assumptions
  - There are no message losses
  - There are no process crashes (**not fault-tolerant**)
    - Failures can be detected with `pings` with timeouts by the **Failure detector**
      - The other node must not reply **for a given number of times** before considering it dead.
      - If the other node is not actually crashed (e.g.: huge network delays so it's ok to consider it dead), a node will keep ignoring future messages coming from that node.
      - $timeout > 2 \cdot T_{transmission\_time} + T_{ping\_processing}$
- Each process can be in 2 states:
  - `PARTECIPATING`
  - `NON-PARTECIPATING`
- Phases
  - Phase 1
    - Initially, all nodes in the `NON-PARTECIPATING` state
    - A nodes wants to start an algorithm run
      - It sets its state to `PARTECIPATING`
      - It sends an `ELECTION` message with its `UID` to its successor
    - A `NON-PARTECIPATING` node, upon receiving an `ELECTION` message
      - Switches its state to `PARTECIPATING`
      - Forwards the `ELECTION` message with the highest `UID` among its own and the one contained in the message
    - A `PARTECIPATING` node, upon receiving an `ELECTION` message
      - Forwards the message if the `UID` contained in the message is greater than its own
      - Ignores it if it is less than its own `UID`
        - There's another election run, hence there's more than one `ELECTION` message
      - **Becomes the leader** if the `UID` contained it's **equal** to its own

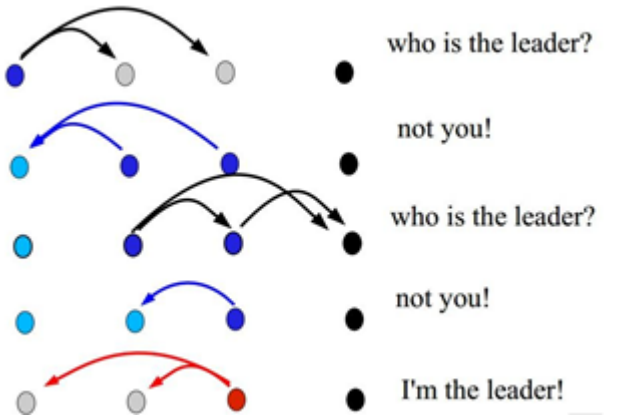
- Sets its state to NON-PARTECIPATING
- Sends a LEADER message containing its own UID
- Phase 2 begins

- Phase 2
  - A non-leader node, upon receiving a LEADER message
    - Sets its state to NON-PARTECIPATING
    - Stores the leader UID
    - Forwards the message
  - The leader node, upon receiving the LEADER message
    - The message is not forwarded
    - The algorithm terminates

- Message complexity: $O(N)$
  - Worst-case scenario: the **algorithm starter is right next to the leader-to-be**
    - Phase 1
      - $N-1$ until the ELECTION message reaches the process with the highest UID
      - The ELECTION message is then forwared N times before returning back to the leader
    - Phase 2
      - N LEADER messages
    - In total, $3N-1$ messages
  - Best-case scenario: the leader-to-be starts the algorithm
    - $2N$ messages, still $O(N)$

Bully algorithm

- Each process knows the UID of each other
- Processes ordered from left to right, with increasing UID



ELECTION message
ANSWER message
ELECTION message
ANSWER message
LEADER message

- The process with the highest UID is the leader, so in this case, the last one in black
- When it crashes (that's why it's black), the other processes start a run of the algorithm upon detecting its failure
  - "*Who is the leader?*": A process sends an ELECTION message to those processes with larger UID than its own
  - Two possible outcomes

        - "*Not you!*": These processes reply with an ANSWER message
            - These processes start an algorithm run
            - The requesting process waits for the leader's ID (light blue state)
        - No reply arrives within a timeout
            - The process is then the **new leader**
            - It warns everybody by broadcasting a LEADER message
    - A process with the largest UID might enter the system: it **imposes** itself as the **new leader** (bully)
    - Message complexity: $O(N^2)$
        - Worst-case scenario: the process with the smallest UID starts an algorithm run.
            - First run:
                - $N-1$ ELECTION messages
                - $N-1$ ANSWER messages
            - Second run:
                - $N-2$ ELECTION messages
                - $N-2$ ANSWER messages
            - At the end: $N-1$ LEADER messages
            - In total: $\frac{2N(N-1)}{2} + N - 1 = N^2 - 1$

Reliable multicast

Tuesday, August 22, 2017
15:55

- `multicast(g, m)`: sending a message `m` to a group g
- Assumptions
    - Reliable send/receive operations
        - Acks are needed
    - Processes may crash
    - Sender belongs to g
- Multicast types
    - Basic multicast (`B-multicast()`, `B-deliver()`)
        - Validity: every correct process in g will deliver the message if the multicaster doesn't crash
    - Reliable multicast (`R-multicast()`, `R-deliver()`)
        - Integrity: delivery is performed *at most once*
        - Validity: if a correct process multicasts a message, it will also deliver it
        - Agreement: if a correct process delivers the message, then all other correct processes in the group **will** eventually deliver it

Implementations

- Basic multicast
    - Built on top of send/receives primitives

	Multicasting	**Delivering**

Pseudo code	• `B-multicast(g, m) {`
	`  // single send operations`
	`  for(p in g) send(p, m);`
		• `B-deliver(m) {`
		`  receive(m);`
		`}`

	} }	
Notes	• Too many acks could arrive: some of them could be dropped, causing retransmission (*akc implosion*) ▪ Messages could be sent following a spanning tree topology ▪ A multicast service provided by the underlying network could solve the problem	

Reliable multicast

- Built on top of the basic multicast

	Multicasting	**Delivering**
Pseudo code	• `R-multicast(g, m) { B-multicast(g, m) }`	• `R-deliver(m) { B-deliver(m); if( /* only if m was not received before (otherwise the multicast would last forever) */ ) { B-multicast(g, m); R-deliver m } }`
Notes		o Basic multicast might crash during its execution, some processes might not receive the message (agreement property not satisfied) • Messages could be re-transmitted by other processes before they deliver them • The last `R-deliver m` pseudo-statement indicates the message delivery completion to the application ▪ Duplicates • Can be filtered

- Message complexity: $O(N^2)$, because each process has to send N messages

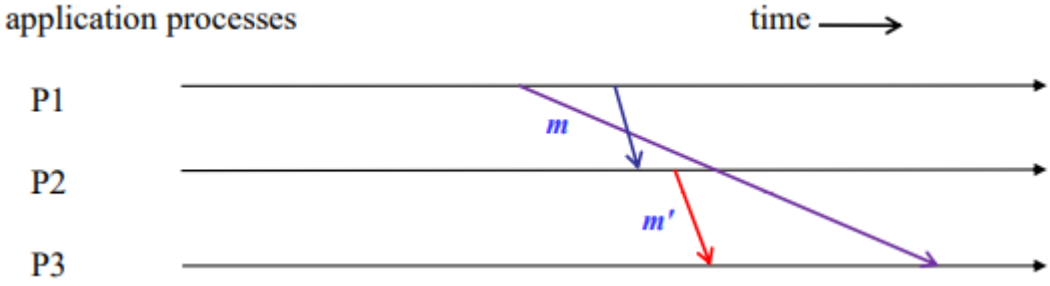
Ordering guarantee

- Ordering semantics
  - Partial ordering
    - Relation between the order in which messages are sent and the order in which they are received
      - Send operations might be concurrent, so there's no ordering constraint at the receiver

- Not all messages are sent by the same process
- Some multicasts are concurrent (not ordered by happened-before)

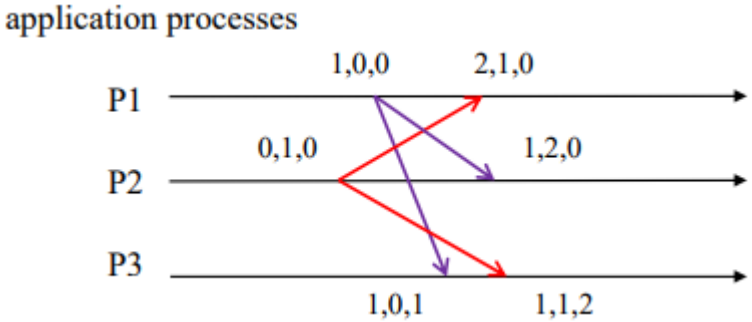
- Types
  - **FIFO** ordering (order within one sender): If a correct process issues multicast(g, m) and then the **same** process issues multicast(g, m'), then every correct process that delivers m' will deliver m before m'.
    - Solution: **hold-back queue** with **sequence numbers**

      ⇑ *since any two multicasts by the same process are related by happaned-before*
  - **Causal** ordering (order within multiple senders): If a correct process issues multicast(g, m) and then **another** correct process in g issues multicast(g, m') (multicast(g, m) → multicast(g, m')), then **any correct process that delivers m' will deliver m before m'**.
    - Violation example with one-to-one message

      P3 delivers m' before it delivers m.

    - Solution**: hold-back queue** with **vector timestamps**
      - Mantained by the message service at each node, for each process

      - The vector has N state counters
      - For each process, each entry record the most-up-to-date value of the **state** counter (updated during both multicast sending and receiving) delivered to the application process, for the process at that position
      - $j \xrightarrow{<message,\ V_j>} i$
        - i stores the message in its hold-back queue until

          The message is exactly sent by j

          $V_j[j] = V_i[j] + 1$

          i already delivered (at least) all messages also delivered by j

          $V_j[k] \leq V_i[k], k \neq j$
        - The message can then be delivered to i

- **Total** ordering (): If a correct process delivers message m before it delivers m', then any other correct process that delivers m' will deliver m before m'.
  - Stronger semantic: it requires messages to be received in the same order by all processes.
  - Solution: **agreed sequence numbers**, unique **all over the group**
    - Each process i keeps track of
      - A_i: biggest **agreed** number observed so far
      - P_i: biggest number **proposed** by itself
    - Upon receiving a new message from p, receiving processes send **proposed** sequence numbers to p, which uses them
    - ...

Consensus

Thursday, August 24, 2017
12:13

- Processed need to **agree** on some value
- They have a **set of proposed values**
  - `findDecision(<list of proposed values>)`
    - Must return the same value for all processes
    - Returns a special value (no decision) in parity cases
  - Building it
    - Each process proposes a value, broadcasts it and keeps track of other proposed values
- It has to be fault tolerant
  - Assumption: system is **synchrounous**
    - Processes take a bounded time to complete a communication step.
      This allows **failure detection**, by looking at processes that don't reply within this bounded known time.
- Status
  - Decided
  - Not decided
- Requirements
  - Termination or liveness: eventually, each correct process sets its decision variable in a bounded period of time
  - Agreement: same decision value for all correct processes
  - Integrity: if all correct processes proposed the same value, then any correct process in the decided state has chosen that value

Solution

- Up to f processes may crash
- N total processes in the system
- Round
  - Initially, processes multicast only their own proposed value
  - Each process multicasts a **new** proposed value everytime a **new one** is received
    - At the beginning, each process' list contains only its proposed value
  - Driven by **timeouts**: since the system is synchronous, each round has a deadline, after which each process that has not responded is certainly crashed.

- The algorithm evolves in $f + 1$ rounds
  - Necessary to cope with crashes
  - Only 1 round is necessary if no process crashes
  - Additional rounds, one for each crashed node, allows processes to exchange their values again, this allowing them to retrieve the missing values (by removing those values which are not received again).

- Algorithm
  - In round r $(1 \leq r \leq f + 1)$:

```
// New values broadcast
 B-multicasts(g, Values_i^r \ Values_i^{r-1});

// Preparing the new set of values
 Values_i^{r+1} = Values_i^r;
while(<in round r>) { // driven by timeouts
      upon receiving a new value V_j from process q
      Values_i^{r+1} = Values_i^{r+1} ∪ V_j; // inserted only if not present
}

// Next round
r++;
```

  - After $f + 1$ rounds:

```
decision_i = findDecision(Values_i^{f+1});
```

  - Consensus is reached
    - $decision_i$ is the same for everyone
    - $Values_i^{f+1}$ is the same for everyone

- Effectiveness proof by contradiction
  - At round $f + 1$, two processes i and j have two different sets of proposed values, $Values_i^{f+1}$ and $Values_j^{f+1}$
  - There might be a value which is present in the second set, but not in the first one
  - This **can happen** if some process k (at least one), during round $f + 1$, succeeded to send that value to process j, but crashed before sending it to i.
  - This means that at the previous round f, process k had this missing value that i and j didn't have.
  - This implies that also at round f at least one crash occured, for the same reason.
  - At round 1, the number of crashes rise up to $f + 1$, which is against the hypothesis of f crashes.

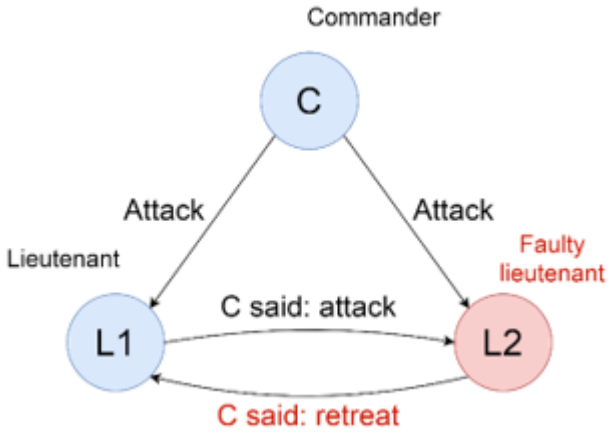
The Byzantine Generals problem

Thursday, August 24, 2017
16:56

- Consensus problem with Byzantine processes
- Faulty processes could send different values to different processes
- Assumption: the system is still synchronous
- Requirements

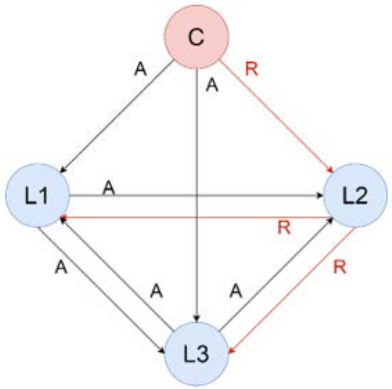
- Termination: eventually each correct process sets its decision variable
- Agreement: when the algorithm terminates, all processes agree on the decision
- Integrity: if the commander is correct, then all correct processes decide on the value that the commander proposed

- Faulty lieutenant example
  - Number of generals (all nodes, commanders and lieutenants) $N = 3$
  - Number of faulty processes $f = 1$
  - Scheme

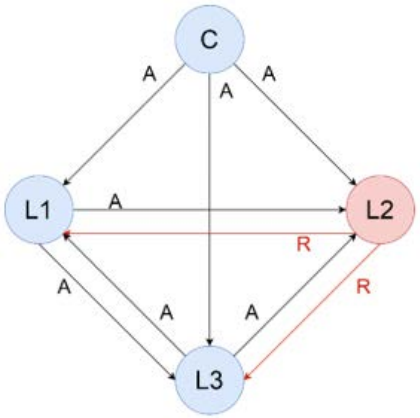


  - L1 cannot understand which message is correct, except for digitally signed messages
  - Conclusion: it's impossible if $N \leq 3f$

- A solution exists for $N \geq 3f + 1$
  - $\begin{cases} N = 4 \\ f = 1 \end{cases}$
  - Two rounds (like the figure above)
    The commander sends a value to each of the lieutenants
    Each lieutenant sends the received value to its peers
  - Each lieutenant receives
    - A value from the commander
    - $N - 2$ values from its peers
      - N – the commander – itself
  - Faulty commander



  - Faulty lieutenant



- In either case, a correct lieutenants only needs to apply a simple **majority function** to the set of values it receives

The FLP restriction

Friday, August 25, 2017
12:22

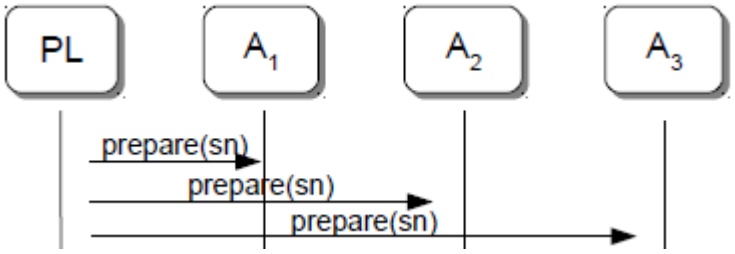
- Consensun problem in asynchronous systems
- **FLP restriction** statement or **impossibility result**: "*It's impossible to find an algorithm that guarantees to reach consensus in an asynchronous system, in a finite expected time, if at least one process may crash*"
  - This doesn't mean that processes can never reach distributed consensus in an async system if one of them is faulty (it says "*guarantees*")
  - In practice, this has been done successfully because processes fail while reaching an agreement quite rarely. Systems are instead almost always **partially synchronous**, instead of asynchronous.
- Practical approaches
  - Fault masking
    - Crashed processes **restart**
    - Every process saves in a persistant storage all the infos permitting it to recover its state and continue its execution
    - Processes just take longer time to respond
  - Failure detection
    - With `pings`
    - A failure must be announced to other processes
    - False positive failures won't be handled, by making the failure *fail-silent*, i.e. by discarding further messages sent by the excluded process
    - Consensus algorithms allow processes to re-join the computation group
  - Randomization
    - If failures are introduced by attackers, randomized messages can avoid this

Paxos

Monday, August 28, 2017
09:26

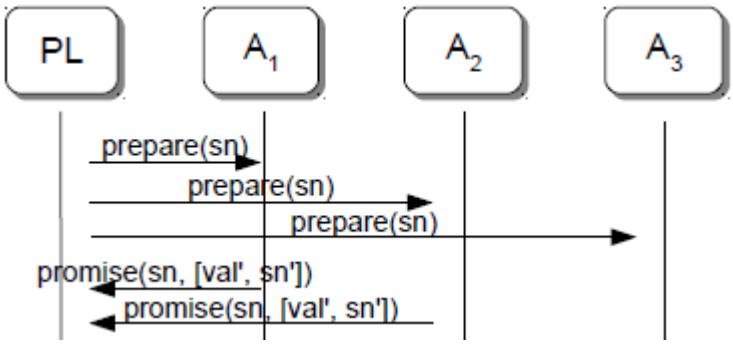
- Fault-tolerant consensus algorithm

    - It allows to update data in eventually consistent systems
    - It's repeated every time there's an update or periodically
- Works with replicated data
- Roles
    - Proposing leader: it proposes new data values
    - Acceptor: it votes to accept a proposed value
    - Learner: it will be informed of the voting outcome
- A process can assume **more than one role**
- Message complexity: $O(5N)$
- Algorithm
    - A proposing leader PL broadcasts a `prepare` request to all the acceptors with a **unique** and **ever-increasing** sequence number (`sn`)

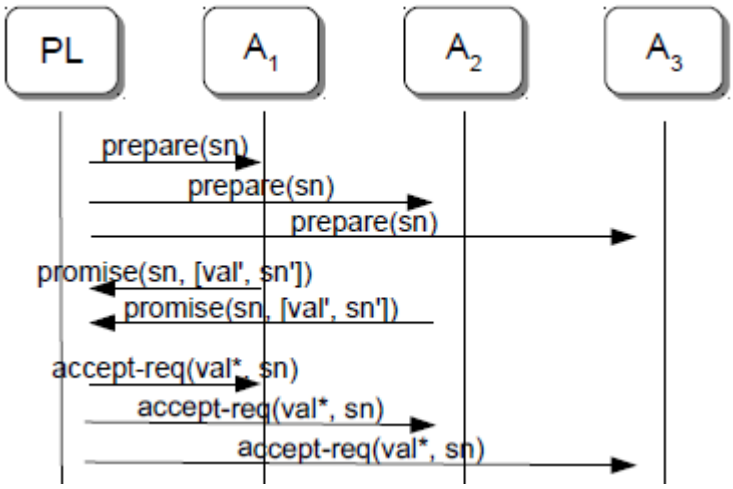


- Acceptors A_i reply with a `promise` message, **promising not to accept any other `prepare` request with lower `sn`** by ignoring them, or even better, by explicitly rejecting them.

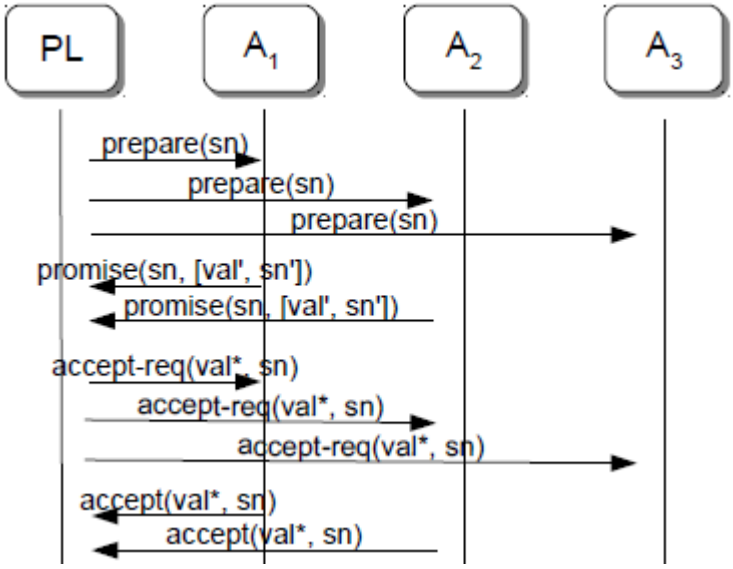
  They will include:
    - The last accepted value `val'`
    - The corresponding sequence number `sn'`



- If the **majority** on acceptors A_i replied and they have not accepted `prepare` requests with higher SN, the proposing leader PL sends an `accept-req` to all acceptors. It includes:
    - The same original SN `sn`
    - The value to be accepted `val*`
        - It might be totally new or equal to `val'`



Acceptors A_i accept with an `accept` message, **iff they didn't reply another prepare request** with higher SN



5. Once the **majority** of acceptors A_i accepted, the proposing leader PL broadcasts a `commit` message.
   - The new value `val*` is now considered **accepted**, and it's **communicated to the learners**

Distributed commit

Friday, August 25, 2017
12:49

Problems

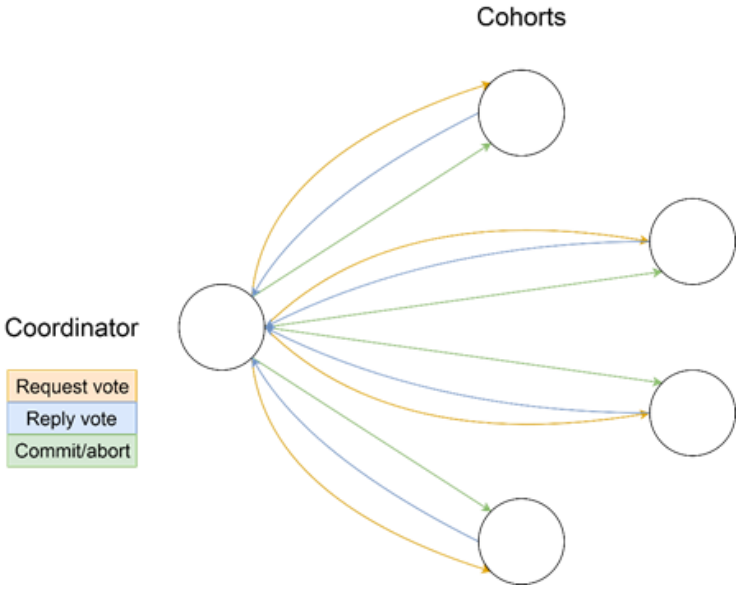
- Faults
- Replica management: process of keeping track of where portions of a (very big) data set can be found

Types

1. Distributed transactions
    - Transactions on data distributed in different locations
    - Nodes must
        - Complete the transaction correctly
        - Agree on the final outcome
        - Agree to commit the transaction, making it permanent
            - Otherwise the transaction should be aborted by everyone
            - This is an agreement problem (commit or abort transaction)
2. Data replication
    - For availability (despite server failures)
    - Requirements
        - Consistency among replicas (edit propagation)
        - Modifications agreement
            - It might happen that different server are serving different clients at the same time (proximity or load balance)
            - Clients may ask for concurrent modifications
            - If modifications are in conlict, one of them may be aborted
                - This is also an agreement problem (commit or abort modifications)

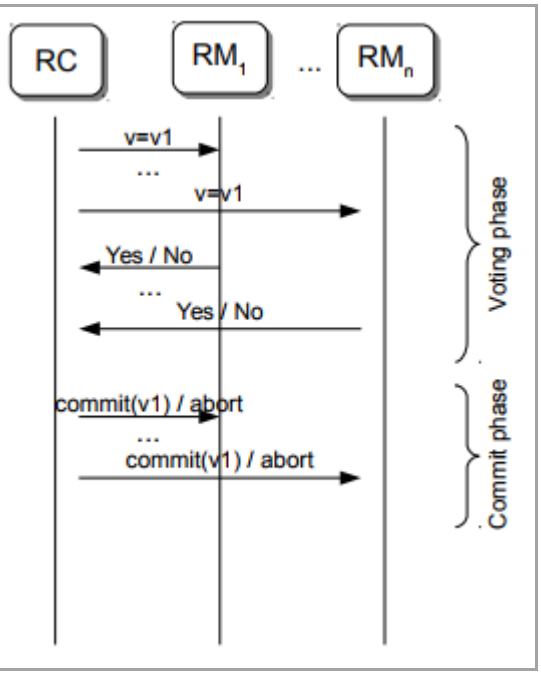
2-phases commit

Friday, August 25, 2017
12:53

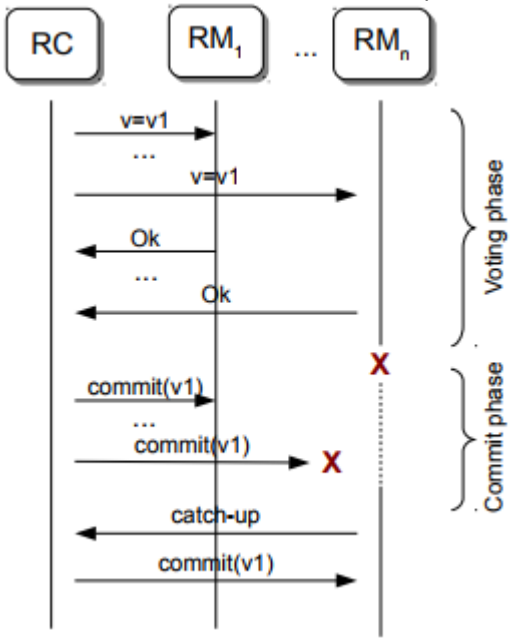


- For commit or abort a transaction (binary consensus)
- Not decentralized
  - Proposers are called **cohorts** (or Resource Managers **RMs**)
  - A **coordinator** (or Resource Coordinator **RC**)
    - Gathers votes from all proposers
    - Decides (commit/abort)
    - Informs all cohorts
- 2 phases

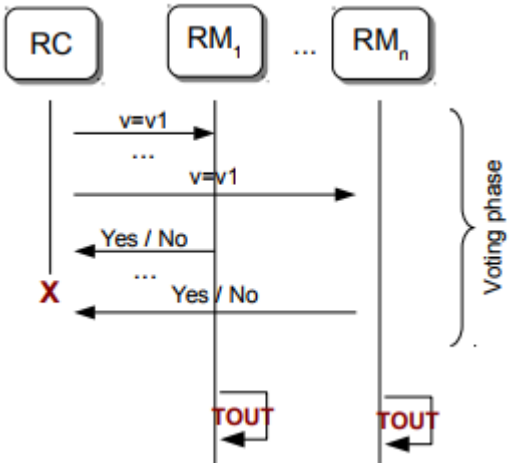
1. Voting phase
   - The coordinator asks a vote (**value agreement**)
   - Cohorts send their vote ("**yes/no**", commit/abort)
2. Decision phase
   - The coordinator checks that everyone voted
   - The coordinator decides and sends the result to everyone (commit if everyone voted yes)



- Message complexity: $O(3N)$
- Requirements
  - Consistency
  - Fault tolerance
    - By a cohort
      - Before voting: coordinator aborts for safety reasons
      - After voting: no action is needed
        - If it comes back, it needs to catch-up the coordinator reply



    - By the coordinator
      - Before receiving all replies: no commit/abort send
        - RMs abort on timeout (TOUT)



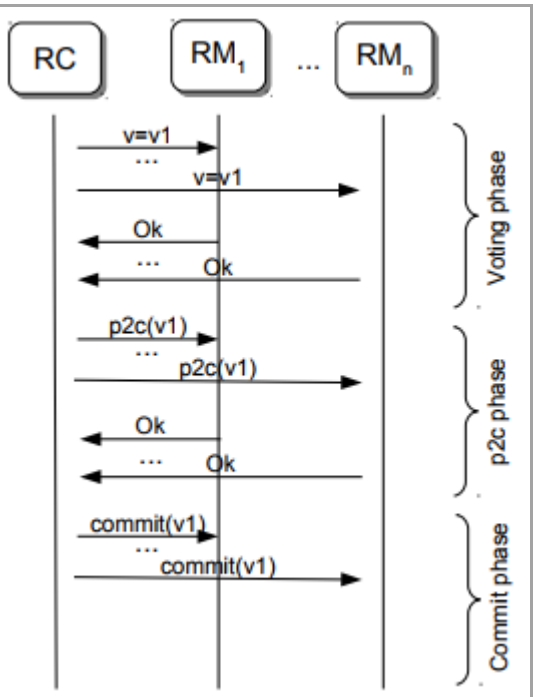
      - While sending the response, some cohorts will keep waiting for it.
        A new coordinator will replace the faulty process thanks to fault masking.
        - This could cause high latency
        - The 3-phases commit algorithm resolves this problem

3-phases commit

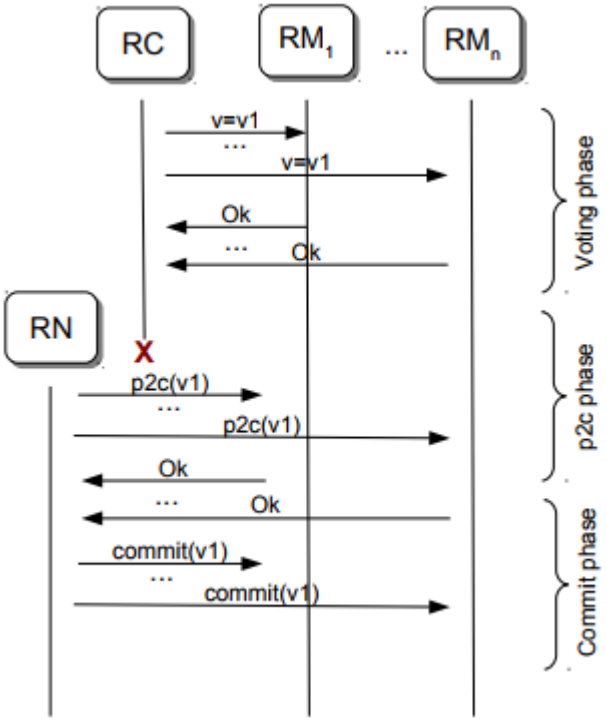
Friday, August 25, 2017
14:33

- It avoids processes to wait endlessly for a response in case the coordinator crashes while sending responses, by adding an intermediate phase. Each cohort will be able to take over as a coordinator.

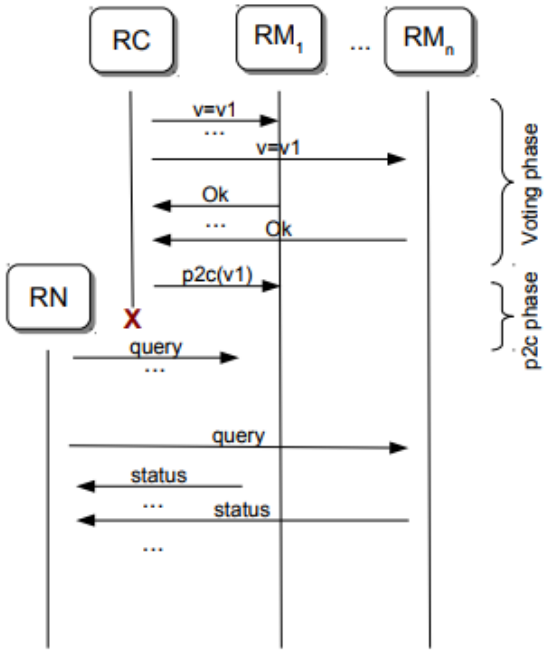
- Voting phase
  1. The coordinator asks for votes
  2. Cohorts reply (yes/no)
  3. The coordinator computes the result (let's assume it's a commit)
- p2c phase
  4. The coordinator sends a `prepare-to-commit` (p2c) message to all cohorts
     - This message informs cohorts about the outcome of vote, but it doesn't allow them to commit yet
  5. When cohorts receive the p2c message, they reply with a `ready` message
- Commit phase
  6. Upon receiving all the `ready` messages, the coordinator orders all cohorts to commit
  7. Cohorts commit once they receive that order



- Advantage: if the coordinator fails after the p2c phase, anybody can replace it with an election
  - This reduces latency
  - Message complexity increases to $O(5N)$
- Fault tolerance
  - By a cohort
    - Before voting: transaction is aborted
    - After voting: no action is needed
  - By the coordinator
    - During the voting phase: another node replaces it
    - During the p2c phase: another node replaces it
      - If the new coordinator had already received a p2c message, it broadcasts the p2c message once again



- Otherwise, the new coordinator will query other cohorts
  - If any cohort has seen a p2c, it will complete the protocol by sending out all missing p2c and commit



  - Otherwise, the new coordinator can either
    - Abort
    - Trigger another vote
- During the "order sending phase": it means that all cohorts have received a p2c message.
  A new coordinator can then be eleceted, and it just has to send orders.

2PC vs 3PC

Monday, August 28, 2017
11:10

2PC

- Tolerates RMs that fail-stop
- Cannot tolerate RC (Resource **Coordinator**) failures

3PC

- Tolerates RC failures
- Less efficient w.r.t. 2PC (5n vs 3n messages)

Both

- Cannot tolerate nodes that fail-recover
- Cannot tolerate network partitions
- All nodes need to answer: bad with large networks with frequent failures

Network partition problem

Friday, August 25, 2017
14:47

What is it

- Nodes cannot distinguish between node failure and link failure
- If a link connecting two network partitions fails, two **separated** subgroups are formed
    - These two subgroups could become inconsisent with each other

Consistency protocols

- Assumption: partition will eventually be repaired
- Modifications made within a partition should guarantee that data won't be in conflict when the partition will be repaired
- Two ways
    - Optimistic approach
        - Nodes within **any** partition can still modify data
        - Conflicts will be resolved later
    - Pessimistic approach
        - **Only one** subgroup (which hold the quorum) is allowed to modify data
        - There will be no conflicts then

CAP theorem and eventual consistency

Monday, August 28, 2017
09:14

CAP theorem statement

- It's not possible to guarantee at the same time
  - **C**onsistency
    - Strong consistency is not mandatory.
      Weak consistency models are used instead, like the eventual consistency one.
  - **A**vailability
  - **P**artition tolerance

Eventual consistency model

- High availability
- They tolerate faults and network partitions
- If no new updates are made to a given data item, all accesses to that item will **eventually** return the last updated value
  - Replicas **tend to converge** to the most updated value
- BASE semantic:
  - **B**asically **A**vailable
  - **S**oft state
  - **E**ventual consistency
- They're **not** ACID
  - **A**tomicity: commit or abort
  - **C**onsistency: always
  - **I**solation: each transaction executes as if isolated
  - **D**urability
- There must be a replica convergence solution

AWS DynamoDB

Properties

- NoSQL service
- Single-digit ms latency
- Guaranteed throughput
- Elastic table growth, no size limit
- Fault (hw and sw) tolerant
- CAP theorem
  ~~Consistency~~ (eventual consistency)
  Availability
  Partition tolerance

Characteristics

- Table are split into **partitions**
- Data model
  - Table
    i. Collection of items
    ii. Composed of attributes

    - Primary key
        - Partition hash key
        - Optional sort key
- At table creation time, the user must specify
    - RCU: Read Capacity Unit
        - **Consistent** read < 4KB
        - 2 **eventually consistent** reads < 4KB
    - WCU: Write Capacity Unit

  R/Ws beyond this limits are considered as errors.
  These values are dynamic.

Operation tasks

- Primary tasks
    - Up = available
    - Healty = satisfying functional & non-functional requirements
- Secondary tasks
    - Root-cause analysis (RCA)
    - Feed information back to developers
    - Operation handling automation

Web services

- Client/server via HTTP
- Web servers run web applications

Multi-tier architecture paradigm: why

- Different layers (on different machines) for different functionalities
- Advantages of deploying different layers on different machines
    - High specializatoin for higher performance and scalability
    - It allows a single machine to save resources for a single functionality
    - Faults won't deprive a system of multiple functionalities (more availability)
    - Maintainability

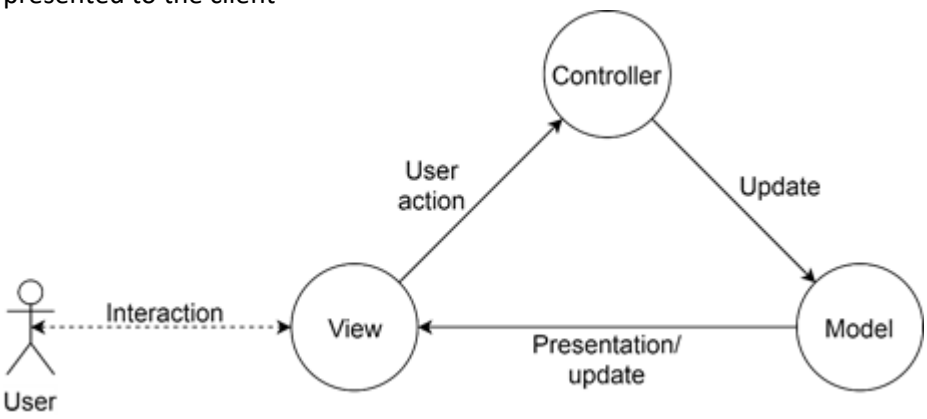
The three-tiers architeture

- Layers
    - Front-end **presentation**
        - Web servers
        - It handles content to be displayed to the client, e.g. web pages
    - Middleware **application logic** or **business**
        - Application servers
        - Contacted by web server: they submit tasks and retrieve results
    - Back-end **data storage** or **persistence** or **data back-end**
        - Database server, DBMS
        - Information storing and retrieval

MVC pattern

What is it

- It stands for ***Model View Controller***
- Used by the *JavaServer Faces* technology
- It's a software architectural pattern that separates the internal data representation from the way it's presented to the client



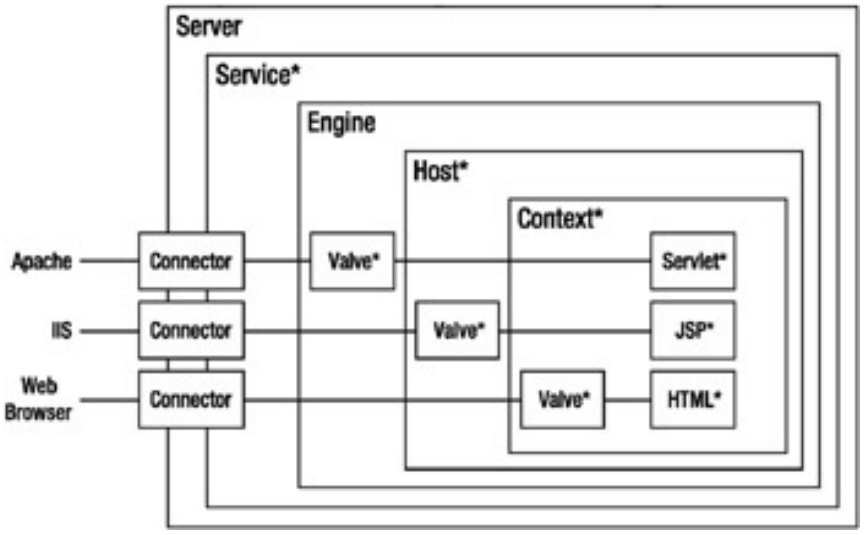
  - **Model**: internal data representation
  - **View**: data user representation
  - **Controller**: it defines *how the user interaction with the view modifies the model*
    - It executes the proper logic to update the model upon receiving commands from the user

Java servlets

Monday, August 28, 2017
11:35

Internal architecture

- **Apache Tomcat** is a web server used for Java web applications
  - It's included in Java EE
- It uses port 8080 instead of 80 by default
- One thread per request
  - All data structures accessed by a Servlet must be thread-safe, since it can be accessed concurrently
- Container-based structure



- This structure is defined by developers in a XML file
  - Contexts, instead, are defined by placing a web application **project directory** in a specific Tomcat directory

Components

- *: there might be multiple instances of it
- Server
  - Tomcat server instance
- Connectors
  - They receive client requests (for instance by HTTP) and turn them into special objects with propers interfaces (like `HttpServletRequest`)
- Service
  - It exposes connectors
  - It receives request objects from connectors
  - It passes these request objects to the engine
- Engine
  - Request-processing
  - It examines HTTP headers to determine to which web application it should pass the request (multiple websites on the same server)
- Host
  - Web application, which contains multiple applications called contexts
  - It allows multiple **virtual servers** to be configured on the same physical machine and to be addressed by different IP addresses or host names
    - In Tomcast, they are addressed by different *fully qualified host names*
    - Different website on the same server
      - Clients' requests will contain both the IP address and the hostname to distinguish them
      - The engine inspects the HTTP header to determine which host is being request
- Context
  - Every host has multiple applications (contexts) having different names
  - It contains the real resourced exposed by the web service
    - Static HTML pages

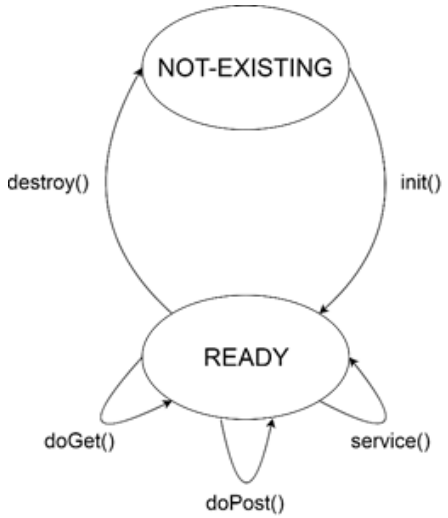
        - Servlets: "*Java program that extends the capabilities of a server, commonly used to implement applications hosted on web servers*"
        - JavaServer Pages (JSP): "*Technology that helps developers create dynamically generated web pages based on HTML, XML, etc.*
          Similar to PHP and ASP, but it uses Java.
          It requires a compatible web server with a servlet container, such as Apache Tomcat or Jetty.".
    - Valves
        - They intercepts requests and process them before they reach their destination
        - Commonly used to log requests

URL mapping

- The `web.xml` file allows to
    - Define servelts
    - Map servlets to names
- Fully qualified name example: http://www.aaa.com:8080/context-path/myServlet
- A Servlet can be mapped to **more than one URL**

Basic API

- Servlets are **special Java objects** used to implement a web service
    - CDS_07_WEBSxaERVER lab
- import javax.servlet
    - The `Servlet` interface must be implemented by the class that will define our Servlet object
        - `init()`
        - The `service(ServletRequest req, ServletResponse res)` method implementation consists in computing a responde `res` for the client
        - `destroy()`
    - Alternatively, a Servlet object can extend the `GenericServlet` abstract class (that still implements the `Servlet` interface, of course)
        - `init()` and `destroy()` are already defined, but they can be overwritten, if necessary
        - The `service()` method still has to be implemented
        - This class is extended by the `HttpServlet` class, which is for HTTP services
            - This is normally used for Servlets
            - The `service()` method needs not to be implemented anymore
            - Extending the `HttpServlet` class forces the developer to implement one callback method per each HTTP request type
                - `doGet(HttpServlet req, HttpServletResponse res)`
                - `doPost(HttpServlet req, HttpServletResponse res)`
                - `doHead(HttpServlet req, HttpServletResponse res)`
                - …
                - They're **callback** methods because they're invoked by the `HttpServlet::service()` method
    - `ServletRequest` and `ServletResponse` are interfaces representing generic request and responses
        - `HttpServletRequest` and `HttpServletResponse` are extensions
- Lifecycle



  - Entrypoint: NOT-EXISTING state
- Accessing information stored in the input `ServletRequest` variable
  - `String getParameter(String name)`
  - `String[] getParameterValues(String name)`
- Writing in the response object `ServletResponse res`
  - `PrinterWriter out = res.getWriter(); // gets an output stream`
  - `out.println(); // write HTML code of the response page`

Sessions

- HTTP is stateless
- It's useful to mantain information about clients
  - They can be mantained thanks to **sessions**
  - It has a lifetime
- The server needs to associate a **session ID** to every client.
  There are more ways to do that:
  - The client can store its session ID and include it in all successive requests, thanks to **coookies**
    - Most used technique
  - With **URL rewriting**, a unique session ID is attached to each URL sent to the client browser
  - The server can put the identifier in a hidden filed of an HTML form, which is returned from the client in the body of its next request
- `HttpSession` interface
  - `req.getSession()` gets the session: it creates a new one if no session exists yet
  - Storing info about the client: `setAttribute(String name, Object value)` method
  - Retrieving info about the client: `getAttribute(String name)` method

JavaServer Pages (JSP)

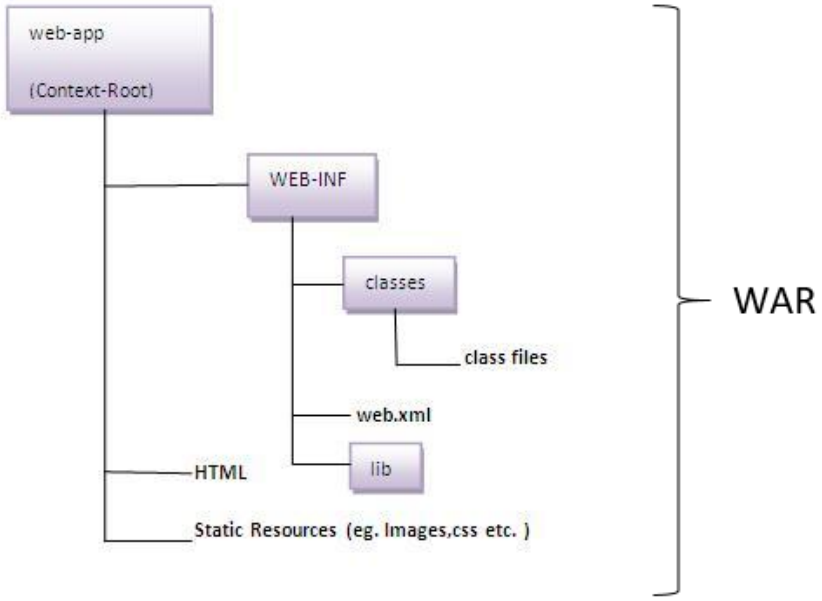
- It would be preferable to decouple HTML from Java code, instead of writing HTML code into a `ServletResponse` object with the `out.println()` method
- JSP is a template system
- JSPs are translated into Servlets when compiled
- JSP is mainly made up by HTML code, with some embedded Java in it

Pros	**Cons**

• Flexible • Maintainable	• Servlets have more control over requests handling

Apache Tomcat: configuration

- Directory hierarchy
  - Folders: during development
  - **W**eb **AR**chive (WAR): during deployment (can be done by NetBeans)



- Document root
  - HTML, JSP files, static content, …
  - For example, the `<document root>/index.html` file can be found on http://www.aaa.com:8080/context-path/index.html
- `<document root>/WEB-INF/web.xml` is the **deployment descriptor**, an XML config file that defines the application structure (hostname, context name, Servlets, resources names, …)
  - Example

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app>
    <!-- General description of your web application -->
    <display-name>My Web Application</display-name>
    <description>
        Short description of the application...
    </description>

    <!--
        Context initialization parameters that can be retrieved in a
        servlet or JSP page by calling:
        String value = getServletContext().getInitParameter("name");
    -->
    <context-param>
        <param-name>myParameter</param-name>
        <param-value>abc</param-value>
        <description>
            Short description of the parameter...
        </description>
    </context-param>

    <!--
        Definition of the servlets that make up our application. Paramete
        can be retrieved by calling:
        String value = getServletConfig().getInitParameter("name");
    -->
```

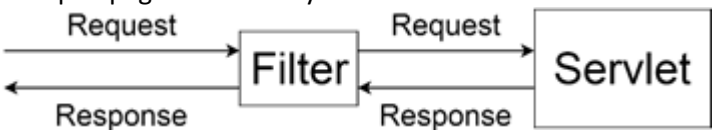
```
    <servlet>
        <servlet-name>FirstServlet</servlet-name>
        <description>
            Short description of the servlet...
        </description>
        <servlet-class>
            mypackage.MyServlet1
        </servlet-class>
        <init-param>
            <param-name>NameParameter1</param-name>
            <param-value>ValueParameter1</param-value>
        </init-param>
        <init-param>
            <param-name>nameParameter2</param-name>
            <param-value>ValueParameter2</param-value>
        </init-param>
    </servlet>
    <servlet>
        <servlet-name>SecondServlet</servlet-name>
    </servlet>

    <!--
        Define mappings between servlets and URLs.
        It is legal to define more than one mapping for the same
        servlet, if you wish to.
        In this example, the FirstServlet is mapped to
        http://hostname:8080/context-path/first
        while the SecondServlet is mapped to
        http://hostname:8080/context-path/*.second
        and the character '*' is a wildcard that matches any substring
    -->
    <servlet-mapping>
        <servlet-name>FirstServlet</servlet-name>
        <url-pattern>/first</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>SecondServlet</servlet-name>
        <url-pattern>/*.second</url-pattern>
    </servlet-mapping>
</web-app>
```

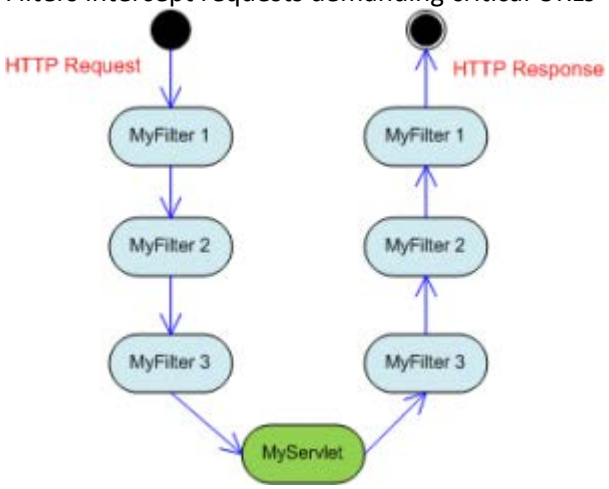
  - A Servlet can be mapped to more than one url by using the * character or by inserting more `<url-pattern>` tags containing various URLs.
- `<document root>/WEB-INF/classes/`: Java class files and Servlets
- `<document root>/WEB-INF/lib/`: libraries
- Web application **deployment**: placing the folders/WAR in a subdirectory into the `webapps` Tomcast folder
- The **context path** is the path from the `webapps` Tomcast folder to the web application document root
- **Tomcast common library folder**: `common/lib/`, contains libraries that need to be shared among different applications
  - It contains the `javax.servlet` and `javax.servlet.http` packages

Servlet filters

- **Interception** is a mechanism to
  - Preprocess requests before they arrive to the Servlet
  - Postprocess responses before they're sent to the client
- Example: pages visible only to **authenticated** users



  - Filters intercept requests demanding critical URLs



    - Actually filters can help any kind of resource, not just Servlets
  - This avoids the Servlet to manage authentication problems
- `Filter` interface
  - `init()`
  - `doFilter(ServletRequest req, ServletResponse res, FilterChain chain)`
    - Implementation example:

```
void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
                                              throws IOException, ServletException {
      // preprocessing code here

      // calls the service() method of the corresponding Servlet
      chain.doFilter(req, res);

      // postprocessing code here
}
```

  - `destroy()`
- Declaring filter in `web.xml`

```
<filter>
    <filter-name>FirstFilter</filter-name>
    <filter-class>mypackage.MyFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>FirstFilter</filter-name>
    <url-pattern>/first</url-pattern>
    <!--
        In this way all the requests directed to the URL
        http://hostname:8080/context-path/first

        It is also possible to map the filter directly to a specific servlet
        by writing:

        <servlet-name>FirstServlet</servlet-name>

        instead than using the <url-pattern> tag
    -->
</filter-mapping>
```

- Filters can be associated with any resource, not just Servlets
- The wildcard * can be used

Java annotations to avoid specifying `web.xml`

- This has to be specified before the Servlet class

```
@WebServlet(
    name="FirstServlet",
    urlPatterns={"/first"},
    initParams={
        @InitParam(name="n1", value="v1"),
        @InitParam(name="n2", value="v2")
    }
)

public class MyServlet1 extends HttpServlet {
    //...
}
```

- Short annotation for URL mapping only (no name and parameters)

```
@WebServlet("/first")
public class MyServlet1 extends HttpServlet {
    //...
}
```

  - Default name will be `MyServlet1`
  - No parameters

- `web.xml` will be automatically produced by the compiler

Java EE

Tuesday, August 29, 2017
10:14

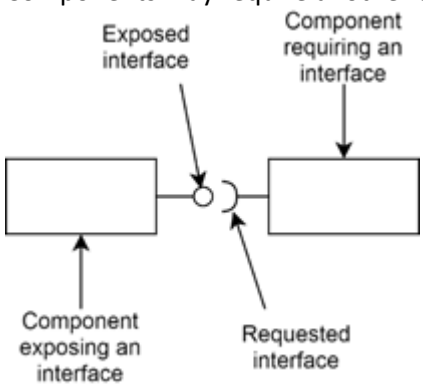
- Java SE + APIs for developing **multi-tier** and **distributed architectures**, **web services** and for **persisent data storage**
- Java EE includes
  - Servlets
  - Enterprise JavaBeans
  - Java Message Service
- Concurrency, distribution/communication, security, data persistency… all transparent

Components

- They are implemented by Java Objects, but:

Object	**Components**
Abstraction of a real-world entity	Unit which takes some responsibilities

- Components may require another component to carry out a service (requested and provided **interfaces**)



- Good for separation of concerns

Application servers

- Java EE runs on application servers
- Application servers are a super-class of web servers

Web servers	**Application servers**
• Reply to clients via HTTP	• Exposes business logic to clients with any protocol, including HTTP

- GlassFish
  - Included in NetBeans + Java EE installation package
  - GUI configurations
    - Access credential
    - Security
    - Database connections
    - Application deployment
    - Add resources like Java Message Service connection factories

EJBs

Monday, September 25, 2017
18:23

- "Enterprise Java Beans"
- Server-side software components that **encapsulate business logic of an application**
  - Components: EJBs are objects with responsibilities
  - EJBs can provide interfaces to other EJBs
- EJBs types
  - **Session** EJBs
    - Triggered on method invocation
  - **Message-driven** EJBs
    - Triggered by a message reception event (Java Message Service)
  - **Entity** EJBs
    - Used to manage data persistence
    - Replace by the Java Persistence API (JPA), which interacts with DBMSs and manages data persistance